UNIVERSITY OF CALIFORNIA

Santa Barbara

Detecting and Preventing Attacks Against Web Applications

A Dissertation submitted in partial satisfaction of the
requirements for the degree Doctor of Philosophy
in Computer Science

by

William Kim Robertson

Committee in charge:
Professor Richard A. Kemmerer, Co-Chair
Professor Giovanni Vigna, Co-Chair
Professor Christopher Kruegel

June 2009

The dissertation of William Kim Robertson is approved.

Professor Christopher Kruegel

Professor Giovanni Vigna, Co-Chair

Professor Richard A. Kemmerer, Co-Chair

June 2009

Detecting and Preventing Attacks Against Web Applications

by

William Kim Robertson

ACKNOWLEDGMENTS

I would like to acknowledge the following people, without whose support this dissertation would not have been possible.

My advisors, Dick and Giovanni, have been a continual source of inspiration and support throughout the years that I have had the pleasure to work with them. Without their guidance, not only would I have not been able to complete the work contained in this dissertation, I likely would not have considered pursuing a Ph.D. at all. For these reasons and more, I am extremely grateful.

To Chris, a special debt of gratitude is owed. Since coming to UCSB as a postdoc, he has been an unflagging fount of knowledge, resolve, and advice, setting an example as a researcher that I have strived, in my own limited way, to emulate.

To all of the past, present, and affiliated members of the UCSB Security Group, thanks for all the great memories. I will never forget the marathon hacking sessions, the inevitable races to submit papers at the last possible second, or the drinking sessions in IV or at the Merc. In no particular order, and with apologies to those that I may have unintentionally omitted, thanks to Darren, Fredrik, Davide, Engin, Marco, Vika, Lorenzo, Fede, Ludo, Luca, Bob, Martin, Max, Brett, Adam, Nick, Sean, Greg, Collin, Wilson, Matt, Dan, Ryan, and everyone else.

`Greetz to the Shellphish and iSecLab crews!`

Thanks to all the people in and around Santa Barbara who made my time here so en-

joyable.

Thanks to my parents and brother for their support during my academic career.

Finally, my special thanks to Jenny for her unconditional love, support, and patience over the years.

VITA OF WILLIAM KIM ROBERTSON

June 2009

RESEARCH INTERESTS

My primary research interest is in protecting web applications from attacks. One line of research has focused on using statistical machine learning techniques to automatically construct models characterizing the normal behavior of web applications in order to perform accurate, black-box anomaly detection of web-based attacks. Recent work in this area has focused on addressing fundamental challenges to performing web application anomaly detection, such as reducing the rate of false positives, adapting to changes in web application behavior over time, and compensating for the scarcity of training data. Another approach I am investigating is the construction of development frameworks to build web applications that are automatically secure against certain classes of attacks by construction.

Other research interests include intrusion detection system testing and evasion, electronic voting security, and the verification of software security properties using static and dynamic analysis techniques.

EDUCATION

University of California, Santa Barbara

Ph.D. Candidate, Computer Science

Santa Barbara, CA USA

June 2003 – Present

University of California, Santa Barbara

B.S., Computer Science

Santa Barbara, Ca USA

September 1997 – June 2002

ACADEMIC EXPERIENCE

University of California, Santa Barbara

Research Assistant, Computer Security Group

Santa Barbara, CA USA

June 2003 – Present

- Advised by Professors Dick Kemmerer and Giovanni Vigna
- Member of Ohio Evaluation and Validation of Election-Releated Equipment, Standards, and Testing (EVEREST) Red Team (ES&S system)
- Member of California Top-To-Bottom Electronic Voting Systems Review (TTBR) Red Team (Sequoia system)

TEACHING EXPERIENCE

University of California, Santa Barbara

Guest Lecturer

Santa Barbara, CA USA

September 2003 – September 2007

- Guest lecturer for introductory computer security course
- Guest lecturer for graduate computer security course

PROFESSIONAL EXPERIENCE

WebWise Security, Inc.

CTO, Co-Founder

Santa Barbara, CA USA

September 2006 – October 2008

- Co-founded, with advisors, web application security company focused on providing solutions for designing, auditing, and protecting web-based applications and services
- Co-developer of *weblock*, a high-speed anomaly-based web application firewall (WAF) designed to detect and prevent both known and unknown attacks against custom web-based applications and services

Sun Microsystems, Inc.

Intern

Mountain View, CA USA

June 1998 - September 2001

- Designed and implemented a testing framework for system controllers deployed in the Serengeti enterprise server platform in conjunction with the Performance Application Engineering (PAE) group

ABSTRACT

Detecting and Preventing Attacks Against Web Applications

by

William Kim Robertson

The World Wide Web has evolved from a system for serving an interconnected set of static documents to what is now a powerful, versatile, and largely democratic platform for application delivery and information dissemination. Unfortunately, with the web's explosive growth in power and popularity has come a concomitant increase in both the number and impact of web application-related security incidents. The magnitude of the problem has prompted much interest within the security community towards researching mechanisms that can mitigate this threat. To this end, intrusion detection systems have been proposed as a potential means of identifying and preventing the successful exploitation of web application vulnerabilities.

The current state-of-the-art, however, has failed to deliver on the promise of intrusion detection. Misuse-based detection systems are unable to generalize to previously unknown attacks for which no signatures exist. In the context of the web, this is especially problematic in light of the wide proliferation of unique, custom-written web applica-

tions. On the other hand, anomaly-based intrusion detection systems seem well-suited for detecting attacks against web applications. Existing anomaly detection techniques, however, have heretofore proven unfeasible due to several factors: unacceptably high false positive rates, susceptibility to evasion, an inability to adapt to changes in monitored applications, and a lack of explanatory power.

In this dissertation, I present WEBANOMALY, an advanced black-box anomaly detection system that accurately detects attacks against web applications with low performance overhead. WEBANOMALY addresses several of the aforementioned fundamental challenges to anomaly detection using a combination of novel techniques. In particular, the relatively high rate of false positives and lack of explanatory power is ameliorated using *anomaly signatures*, a technique for clustering related anomalies and classifying the type of attack they represent. The problem of local training data scarcity is addressed through the use of *global knowledge bases* of well-trained profiles collected from other web applications. Changes in web application behavior over time, known as *concept drift*, are addressed by treating the web application itself as an oracle of legitimate change. Finally, a novel framework for developing web applications that are secure by construction against many common classes of attacks is presented.

TABLE OF CONTENTS

LIST OF FIGURES

List of Tables

Chapter 1

Introduction

The World Wide Web has evolved from its humble beginnings at CERN in 1991 to become the predominant service of the contemporary Internet.[1] With a currently estimated size of approximately 50 billion unique pages as indexed by the major search engines [21], what was originally a system for serving an interconnected set of static documents is now a powerful, versatile, and largely democratic platform for application delivery and information dissemination. In addition to hosting static documents, the web now enables entities, from individuals to governments to multi-national corporations, to publish complex services and applications to an international audience of staggering size; one survey currently places the number of web users in excess of 1.4 billion, or approximately 21% of the world population [86]. The World Wide Web has connected the world in a way scarcely imaginable a generation ago, and is, one could argue, fundamentally responsible for a radical, ongoing reshaping of society, the full

[1] Indeed, in the eyes of many, the Internet and the World Wide Web are synonymous.

Year	Web-related	Total	Proportion
1999	235	1,573	14.94%
2000	334	1,233	27.09%
2001	502	1,564	32.10%
2002	995	2,419	41.13%
2003	521	1,566	33.27%
2004	1,071	2,762	38.78%
2005	2,464	4,878	50.51%
2006	4,581	7,238	63.29%
2007	3,614	6,726	53.73%
2008	3,915	6,816	57.44%
2009	725	1,808	40.10%
	1,740	*4,339*	
Total	18,957	38,583	49.13%
	19,612	*41,114*	

Table 1.1: Web-related vulnerabilities per year [120]. Projected figures for 2009 are in italics.

ramifications of which are still not completely understood.

That which can be used for good can also be turned to evil, however. With the web's explosive growth in power and popularity has come a concomitant rise in both the number and magnitude of web-related security incidents. Table 1.1 presents a summary of web-related vulnerabilities contained in the Common Vulnerabilities and Exposures (CVE) database. The data contained therein clearly indicates a dramatic increase in both the number of reported web-related vulnerabilities as well as the proportion of all reported vulnerabilities. Though an apparent downward trend in web-related reports was observed in the years following 2006, one must consider that the number of reported vulnerabilities is not necessarily correlated to the aggregate severity of said vulnerabilities.

Organization	Records	Data stolen
TJX	94,000,000	Customer records
CardSystems, Inc.	40,000,000	Credit card records
Auction.co.kr	18,000,000	Customer records, including credit card numbers
TD Ameritrade	6,300,000	Customer records
Chilean government	6,000,000	Personal records, credit card numbers
Data Processors Intl.	5,000,000	Credit card records
UCLA	800,000	Personal records, including SSNs
Oak Ridge National Lab	12,000	Records of visiting researchers, including SSNs

Table 1.2: Sample of known data breaches [92, 93].

A major factor in the increasing scrutiny web-based applications have received from attackers is their popularity. Popular web applications can easily receive millions of unique visitors per day. Therefore, if an attacker is able to inject malicious code into such an application, such that the malicious code attempts to exploit client-side vulnerabilities in browsers or browser extensions, an attacker can infect a large population of end-points in a relatively short amount of time with code of their choosing. This malicious code, once installed on a client machine, can perform a variety of nefarious tasks, such as mining the client for sensitive data, or joining the client machine into a botnet [118, 117].

In addition, unauthorized disclosures of confidential data due to vulnerabilities in web-based applications or services are occurring on an increasingly massive and unprecedented scale. Neither government nor industry have proven immune to data breaches, as Table 1.2 indicates. The aggregate statistics on confidentiality breaches are even more troubling, however. Table 1.3 presents an estimate of the number of personal records released due to unauthorized information disclosures per year. In particular, the number of network-related incidents and individual records exposed versus the total number of

Year	Incidents (Net.)	Incidents (Total)	Proportion	Records (Net.)	Records (Total)	Proportion
2000	5	5	100.00%	371,400	371,400	100.00%
2001	9	9	100.00%	157,250	157,250	100.00%
2002	3	3	100.00%	4,960	4,960	100.00%
2003	7	12	58.33%	5,620,300	6,405,300	87.74%
2004	11	22	50.00%	873,900	32,073,900	2.72%
2005	72	150	48.00%	43,483,986	56,116,124	77.49%
2006	137	514	26.65%	9,351,267	50,990,225	18.34%
2007	141	475	29.68%	10,2437,582	164,471,350	62.28%
2008	179	603	29.68%	19,565,359	83,983,070	23.30%
2009	36	170	21.18%	388,987	5,117,359	7.60%
	86	*408*		*933,569*	*12,281,661*	
Total	600	1,963	30.57%	182,254,991	399,690,938	45.60%
	650	*2,201*		*182,799,573*	*406,855,240*	

Table 1.3: Data loss incidents per year [93]. Projected figures for 2009 are in italics. We note that the apparent discontinuity between 2002 and 2003 is a direct result of stricter regulations mandating the disclosure of privacy breaches enacted by the State of California in 2003.

incidents and records exposed is shown. Though the proportion of network-related disclosures trends downward over time due to an increase in physical attacks, a significant portion of incidents were network-related, and a majority of records were lost due to network vulnerabilities. In addition, this data only pertains to records disclosed from entities based in a handful of countries where strict regulations regarding data disclosure are in force. Soberingly, no records exist for the majority of countries under more relaxed regulatory schemes.

The economic and personal impact on the victims of data breaches is difficult to estimate; the only certainty is that it is immense. The industrial and government sectors have begun to respond by mandating stricter regulatory policies regarding the handling of sensitive data. Yet, despite 43 US states having enacted legislation requiring customers to be notified of data breaches, researchers have found no statistically significant reduction in the number of breaches as a result [108].

One particularly influential example of industry self-regulation is that of the Payment Card Industry Data Security Standard (PCI DSS) [98]. Currently at version 1.2, the PCI DSS is a set of twelve requirements for the handling of credit card information that is jointly published by the major credit card companies. In order to process credit card information for these companies, both merchants and credit card processors above a certain size threshold must periodically demonstrate compliance with these requirements. The PCI DSS groups its requirements into six categories: Build and Maintain a Secure Network, Protect Cardholder Data, Maintain a Vulnerability Management Program, Implement Strong Access Control Measures, Regularly Monitor and Test Networks, and Maintain an Information Security Policy. Of particular interest in the context of this dissertation are Requirements 6.6 and 11.4, which states the following:

> 6.6 *For public-facing web applications, address new threats and vulnerabilities on an ongoing basis and ensure these applications are protected against known attacks by* either *of the following methods:*
>
> - *Reviewing public-facing web applications via manual or automated application vulnerability security assessment tools or methods, at least annually and after any changes*
> - *Installing a web-application firewall in front of public-facing web applications*
>
> 11.4 *Use intrusion-detection systems, and/or intrusion-prevention systems to monitor all traffic in the cardholder data environment and alert personnel to suspected compromises. Keep all intrusion-detection and prevention engines up-to-date.*

Though well intentioned, as we shall argue in Section 1.2, this is easier said than done. Regardless, it is clear that, given the magnitude of the problem, there is much interest in both the security community as well as stakeholders in mitigation of the threats to web-based applications. To that end, the first step is naturally to identify the factors that contribute to the existing web security crisis.

1.1 The state of web security

The web suffers from a number of novel threats in addition to the traditional classes of software vulnerabilities. To place these in context, this section will first present a high-level outline of the architecture of the web as it exists today.

1.1.1 The architecture of the web

The web is a client-server network architecture in which web clients and web servers exchange information using the Hypertext Transfer Protocol (HTTP) over TCP/IP. A web client may be a web browser such as Mozilla Firefox or Microsoft Internet Explorer, or an automated "spider" that traverses the web to, for instance, build a search engine index. A web server hosts a set of web resources organized as a tree, each of which is identified by at least one path from the web server's directory root; popular examples include the Apache HTTP Daemon or Microsoft Internet Information Services. One or more affiliated web servers comprise a web site. Web resources may be static text files, Hypertext Markup Language (HTML) documents, media files such as

images or music, client-side code, dynamic scripts comprising a web application that may output any of the above, or any of a number of other possibilities.

A typical HTTP session proceeds as follows. A web client requests a resource from a web server by issuing one of a number of HTTP client commands. The request specifies the path to the resource, various information contained in request headers, and a set of parameters in key-value format. The web server processes the request and returns a response containing a status code indicating the result of the request. The request may be successful, in which case a response body is returned containing the requested resource. Alternatively, the server may direct the browser to issue a subsequent request to another resource or indicate that an error has occurred. Other resources associated with the original resource, such as embedded images or client-side scripts, may be subsequently requested, not necessarily from the same web server. For full details, please refer to the HTTP/1.1 specification [33]. A sample HTTP request and response pair sent between a web client and server is shown in Figure 1.1.

HTTP cookies

Web servers may store small amounts of data, called cookies, at individual web clients.[2] A cookie is stored by sending a special HTTP response header to the client containing a cookie identifier, value, and a DNS domain specification, possibly with other optional information. Subsequent requests issued to web servers that match the domain specification will include a copy of the cookie unless the cookie expires or is explicitly unset

[2]Cookies are not included in the HTTP/1.1 standard, but rather evolved out of an informal specification produced by Netscape [90].

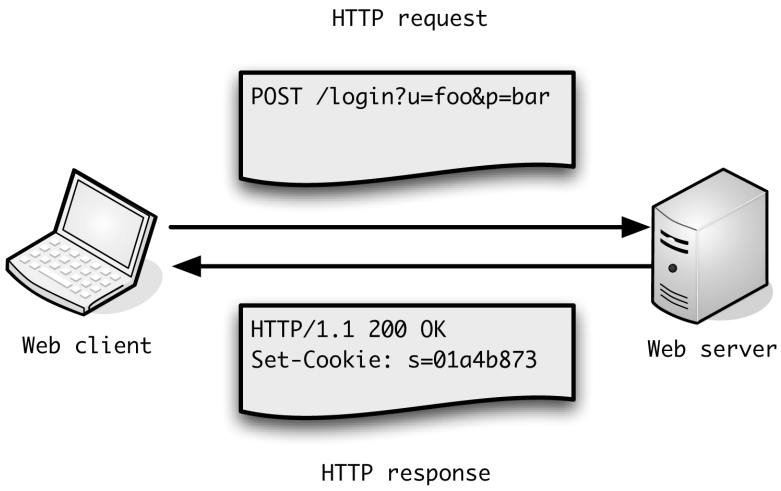


Figure 1.1: HTTP request and response pair.

by the web server. Since HTTP is a stateless protocol, cookies are typically used by web applications to track requests from an individual web client in order to implement an HTTP session. Cookies also form the basis of most web authentication schemes currently in use.

Client-side scripting

Client-side scripting languages such as JavaScript and, later, ECMAScript, gradually became popular as the complexity of the web increased. Executing within the web browser, client-side scripts allow web developers to interact with the Document Object Model (DOM), performing actions such as automatically redirecting the browser to new resources, accessing the browser history, opening new windows, or validating HTML form field content prior to submitting a request to the server. With the inclusion of the `XmlHttpRequest` API and the popularization of Asynchronous JavaScript and

XML (AJAX), client-side scripting has assumed a central role in the development of modern web applications. Using this API, client-side scripts can issue requests that asynchronously update an HTML document within the web browser without initiating a full HTTP resource request cycle to refresh the entire document. This has significantly enhanced the appearance and functionality of web applications, to the point that AJAX-enabled applications have since been collectively referred to as "Web 2.0."

Rich Internet applications

Another significant component of the modern web is rich Internet applications (RIA), such as Adobe Flash or Microsoft Silverlight. In the context of the web, RIA frameworks are used to implement complex client-side applications that display advertisements, stream video, or entirely supplant the HTML document, providing a media-rich, highly interactive environment that could not otherwise be realized. These frameworks typically are developed in modern, high-level languages; for instance, Flash applications are written in a variant of ECMAScript called ActionScript, and Silverlight applications can be written in any language supported by the .NET runtime. These applications are compiled down to bytecode, optionally packaged with media, and executed within a virtual machine runtime available as a plugin for most web browsers.

Security extensions

As HTTP has matured, several extensions intended to bolster its security have been adopted. HTTPS is a combination of HTTP transmitted over a connection that is en-

crypted at the network stream level using the Secure Sockets Layer (SSL) or Transport Layer Security (TLS) standards. Designed to provide end-to-end security to prevent intermediary attackers from observing HTTP communication in transit, it suffers from the drawback that it does not prevent the exploitation of vulnerabilities at either the client or the server, where the vast majority of web attacks occur. In addition, several client authentication schemes have been introduced, but these have also proven ineffective at preventing most classes of web attacks.

Perhaps the most important, and controversial, HTTP security feature is the same-origin policy. First introduced by the Netscape Navigator 2.0 browser, the same-origin policy dictates that client-side scripts executing within the browser may not access resources from other origins, where "origin" is defined to be a DNS domain name, protocol, and network port triplet. This coarse-grained security policy is intended to ensure that only client-side code that has been issued by the web site administrators or developers should execute. This policy has attracted much criticism from developers who find it too restrictive. Yet, bypassing the same-origin policy is the basis of virtually all web client vulnerability classes.[3]

Therefore, several proposals to increase the granularity of the same-origin policy have been introduced. For instance, Flash implements an extension to the same-origin policy with its `crossdomain.xml` specification. These files enable a web server administrator to explicitly declare a set of trusted domains, as opposed to the implicity policy specified by same-origin. Each of these trusted domains can serve Flash applications to clients that can access resources located on that server. The client-side Flash runtime

[3]Thus, one questions whether basing script execution authorizion on the DNS is the correct abstraction.

is responsible for enforcing the declared policy. `crossdomain.xml` has the effect of relaxing the same-origin policy, granting greater flexibility to Flash applications. Concerns have been raised, however, over the difficulty of accurately modeling complex trust relationships. Also, `crossdomain.xml` is specific to Flash applications and does not mitigate threats posed by other types of client-side code.

Other refinements of the same-origin policy have recently been proposed, most notably Mozilla's Site Security Policy (SSP). SSP is intended to address several vulnerability classes arising from the same-origin policy by allowing fine-grained policies to be defined in HTTP headers. These policies enable web server administrators to specify a whitelist of domains a browser should allow as legitimate sources of client-side scripts associated with a specific resource, as well as control how a web server handles cross-site requests. Such proposals are at an early stage at the time of writing, however, and it is unclear what their ultimate effectiveness will be.

Unfortunately, existing web security mechanisms have proven inadequate to the task of protecting web clients and servers from exploitation. As a consequence, the web is plagued not only by the traditional set of vulnerability classes, but, in addition, a novel set of attacks which are not as well understood and for which defense mechanisms are not as advanced. The following sections outline several significant classes of threats.

1.1.2 Web attacks

Attacks against the same-origin policy comprise the majority of novel vulnerability classes in the context of web security. Accordingly, it is essential to recognize the un-

derlying trust assumptions implied by the same-origin policy. These trust assumptions are:

1. The client trusts that resources located within the same domain as the resource originally requested are legitimate.
2. The server trusts that requests from an authenticated client with an active session are legitimate.

The following enumeration of vulnerability classes will elaborate on how abuse of these trust assumptions enables successful exploitation.

Cross-site scripting (XSS)

Cross-site scripting (XSS) attacks are among the most prevalent attacks against web applications [94], and manifest themselves in several ways. A first-order, or reflected, XSS attack proceeds as follows. The attacker coerces a web client to submit a specially-crafted HTTP request to a vulnerable web server, where a malicious client-side script is directly embedded into the request. The response that is returned to the victim client contains the client-side code specified by the attacker, therefore causing the victim to execute malicious code with the privileges of the web server. This code typically attempts to perform an action such as submitting a session authentication token in the form of a cookie to another web server under the control of the attacker.

Another variation on this theme is a second-order, or stored, XSS attack. In this scenario, the malicious script, or a reference to a malicious script, is directly submitted as

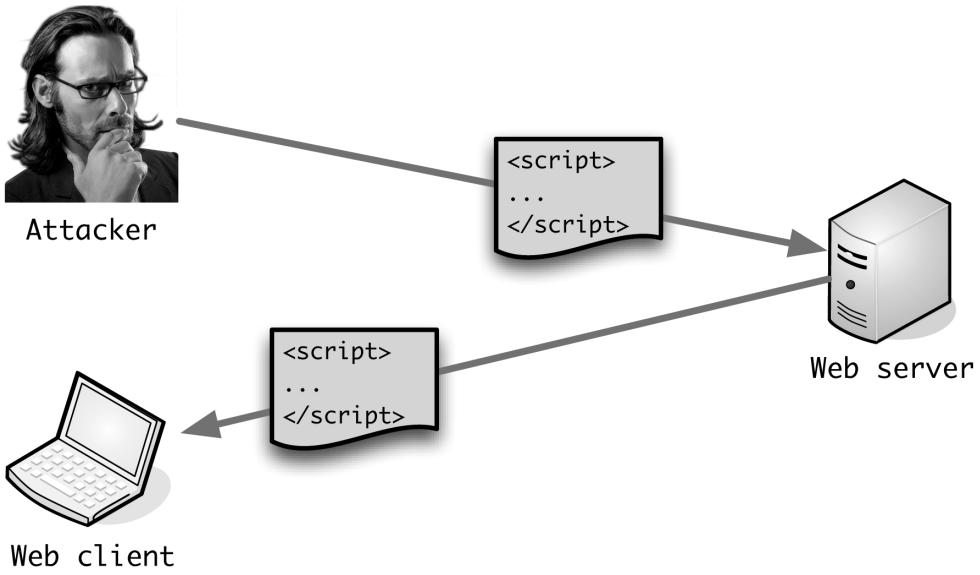


Figure 1.2: Example of a cross-site scripting attack. Here, an attacker has injected a malicious script into a vulnerable web application. The script attempts to exploit a browser vulnerability when executed by a victim.

part of a request by the attacker. The script is then stored on the site (e.g., in a back-end database) by the web server, under the assumption that the stored script will later be included as part of a response to a victim. When the web content including the malicious script is served to a victim, the malicious code is executed as in the first-order case. An overview of a stored XSS attack is depicted in Figure 1.2.

A third, and less frequent, type of this attack is DOM-based XSS. This attack differs significantly from reflected or stored XSS in that it is possible to execute without injecting client-side code into the web application at all – that is, it can be executed completely locally within the victim's web browser. The attack relies upon the existence of client-side code served by the web browser that dynamically updates a document within the web browser through the use of DOM functions such as `document.write()`. For

instance, an attacker may coerce a victim into accessing a URL with an anchor component that contains a malicious `<script/>` reference. When the browser processes this URL, it does not send the anchor to the server, as that information is only required locally. Then, a script embedded into the document returned by the web server dynamically updates the document, inadvertently inserting the malicious `<script/>` node. This node then downloads and executes a malicious script that performs the rest of the attack.

Central to this class of attack is the exploitation of the trust relationship between the web client and server. The client trusts the server not to include malicious code in response to its requests. Due, however, to the failure on the part of the web server to prevent injection of a malicious script by an attacker, this trust assumption can be violated, and the same-origin policy bypassed.

Cross-site request forgery (CSRF)

Cross-site request forgery (CSRF) attacks, in contrast, target the second trust assumption implied by the same-origin policy. CSRF attacks operate as follows. The attacker first coerces a web client to request a resource from a web server under the control of the attacker. Unbeknownst to the victim, the resource that is returned then induces the victim to submit a malicious request to another web server by, for example, including an HTML link or form with that web server as a target. The victim is assumed to have an active HTTP session with the target web server. Hence, the result is an authenticated request to the target web server that can perform arbitrary actions under the control of the attacker. Examples of common actions include modifying the authentication cre-

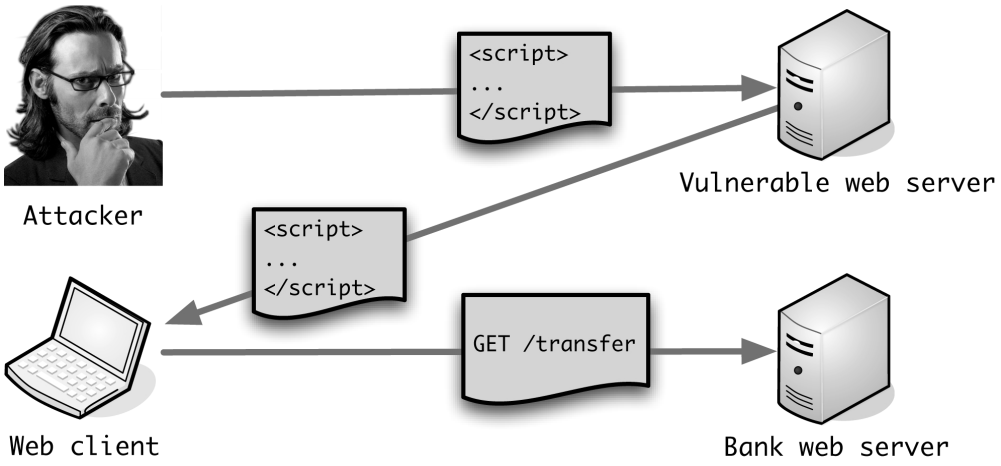


Figure 1.3: Example of a cross-site request forgery attack. Here, an attacker has coerced a victim into submitting a funds transfer from the victim to an account controlled by the attacker.

dentials of the victim to enable the attacker to impersonate the victim, or perhaps, in the case of a banking web site, to transfer funds from the victim's account to an account controlled by the attacker. An illustration of a CSRF attack is presented in Figure 1.3.

CSRF attacks exploit the second trust assumption underlying the same-origin policy, namely that web servers trust authenticated clients to issue legitimate requests. Since, however, an authenticated web client can be deceived by an attacker into issuing malicious requests, the same-origin policy can again be bypassed.

HTTP header injection

HTTP header injection is a vulnerability class that encompasses several variations; examples include HTTP response splitting and HTTP request smuggling. Regardless, each individual attack exploits the same vector, by injecting malicious data into the

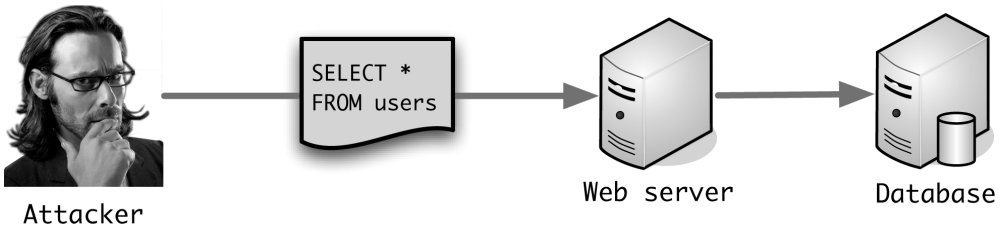


Figure 1.4: Example of a SQL injection attack. Here, an attacker has injected a query that results in the unauthorized disclosure of all user records from the database.

HTTP headers of a request or response. As an example, we consider a second-order HTTP response splitting attack. Here, an attacker first submits a request containing a malicious payload to a web server. Similar to a second-order XSS attack, it is assumed that the payload will later be returned to a victim client as part of a subsequent response. The difference, however, is that instead of being located in the response body, the malicious payload will be located in a response header. The payload can then inject an arbitrary HTTP response under the control of the attacker. This is accomplished by terminating the current header with a "\r\n" character sequence, followed by a set of HTTP headers and arbitrary response body.

In the case of the HTTP response splitting attack described above, it is the trust placed in the web server by the client that is violated. Other HTTP header injection attacks, however, exploit the trust placed in the web client by the server; HTTP request smuggling attacks against intermediary devices such as HTTP proxy servers are one instance.

SQL injection

SQL injection is an exception to this attack enumeration in that it is not a class of attacks that is specific to the web. Nevertheless, due to the widespread usage of databases as a backing store for web applications, SQL injection has comprised a significant proportion of web application vulnerabilities [94]. As such, it deserves mention.

To understand the attack vector, we first describe the Structured Query Language (SQL). SQL is used to submit queries to a database, where each query may perform one or more operations. Common operations include `INSERT`, to insert data into the database; `UPDATE`, to update existing data; `SELECT`, to retrieve data from the database; and `DELETE`, to remove data from the database. A query may require arguments that specify the data to be inserted, updated, returned, or removed; these arguments are typically delimited by single quotes.

SQL injection can occur when data submitted to a web server is used as an argument to a SQL query without proper sanitization. In the simplest type of injection, an argument to a query is allowed to contain the argument delimiter. The effect is to terminate the argument, allowing the attacker to specify the rest of the query. In the worst case, this can lead to enabling an attacker to execute arbitrary SQL queries against the database. Clearly, this can allow an attacker to acquire unauthorized access to data. Since, however, SQL databases often store authentication credentials, SQL injection attacks are often used to bypass a web application's authentication scheme by, for instance, adding a new user account with a known password.

Traditional attacks

In addition to the aforementioned attacks, web clients and servers have also proven susceptible to more traditional types of vulnerabilities. For completeness, we briefly enumerate them in the following.

Both web clients and web servers contain vulnerabilities allowing for successful control flow hijacking attacks. This class of attack generically refers to any attack that allows an attacker to assume control over a program. Well-known examples of this include stack-based buffer overflows that overwrite a saved instruction pointer or otherwise control stack frames; heap-based buffer overflows that enable an attacker-controlled memory overwrite; format string vulnerabilities, enabling an attacker to enumerate memory and perform memory overwrites; and generic pointer overwrites, enabling an attacker to control the destination of a memory write or the target of an indirect function call. Vulnerabilities that allow this class of attack are extremely serious in that an attacker can perform arbitrary actions with the privilege level of the exploited program. Common actions include the installation of malware, or the exposure of confidential data.

Web clients and servers have also been vulnerable to command injection attacks. In particular, web applications that execute external programs during request processing without properly sanitizing any client-supplied arguments have proven to be a popular attack vector. Similar to the previous case, these vulnerabilities are considered to be serious, as arbitrary actions can be performed.

A final example of the traditional attacks that have also manifested themselves in the web context is path traversal. Path traversal attacks typically exploit a vulnerability in a

web server or application that allows an attacker to specify a request for a resource that should not be served. Examples of this attack include escaping a web server document root by accessing its parent directory, or supplying to a web application an absolute path for a file to download instead of an expected relative path.

1.1.3 Web "culture"

In addition to the more technical considerations of web-based vulnerabilities, several other factors that can best be described as cultural characteristics of the web have contributed to its insecurity. First, it is essential to recognize that the presence of vulnerabilities in software alone is insufficient to prompt the enormous number of security incidents that has been observed. Instead, web-based services and applications are mainly targeted due to their status as arbiters of extensive amounts of sensitive information. As was previously demonstrated, criminals can, by exploiting web-based vulnerabilities, acquire vast numbers of personal records, including credit card numbers and social security numbers. This information, in turn, can then be directly used to commit traditional acts of fraud or identity theft. Alternatively, this information can be sold to other criminal organizations, forming one part of an underground economy that has become prodigious in its scale. The availability of sensitive data is, however, fundamentally a consequence of the predominance of the web as a platform for collecting and disseminating information.

Another contributing factor to the insecurity of the web is the relatively informal design methodology. One aspect of this phenomenon is the lack of any real centralized

system architect responsible for its design. This has had positive consequences: anyone with the requisite technical sophistication may contribute to its design, and, as a result, the web has evolved in power and flexibility more quickly and unexpectedly than anyone had imagined. On the other hand, without a visionary "benevolent dictator," new web technologies or development methodologies have often relegated security issues as afterthoughts, or failed to consider them at all.

A characteristic example is that of the same-origin policy and, for lack of a better term, "Web 2.0." Exemplified by the use of AJAX and so-called "mashups," Web 2.0 has resulted in highly composite web pages composed of resources from a multitude of disparate domains. Many of the constituent parts of a page are not under the control of any single entity and may include malicious content that, due to the same-origin policy, is considered trusted. Clearly, the lack of security-aware design that is endemic to the web has resulted in systems with poorly understood security properties that were only recognized after the systems were already deployed.

Yet another cause of web vulnerabilities has been the unfortunate prominence of insecure development tools. While any development tool can be used to create insecure software, it remains true that some tools are naturally more conducive to this than others. As a case study, we shall briefly consider PHP.[4] A server-side scripting language, PHP has become very popular as an easy, powerful language for creating dynamic web sites. Regrettably, however, PHP has also become known for the ease with which insecure code can be produced. Besides a number of implementation-level vulnerabilities that have been discovered, PHP is notorious for incorporating language features that are

[4]PHP was originally an acronym for Personal Home Page, but is now recursively defined as PHP: Hypertext Processor.

either prone to exploitation or difficult to use in a secure fashion. Well-known examples include the `register_globals`[5] and `magic_quotes_gpc`[6] mis-features, among others.

A final cause of the insecurity of the web is the lingering perception that developing web applications is easy. Exacerbated by the availability of development environments like PHP, this perception has resulted in a large pool of practicing web developers unversed in basic security principles. In conjunction with the significant amount of custom, web site-specific development that is typically performed when writing a web application, it is therefore unsurprising that vulnerable software is produced.

In summary, web-based vulnerabilities are pervasive for many reasons, and exploitation of these vulnerabilities has resulted in massive economic damage to individuals, governments, and industry. As a consequence, web-based vulnerabilities pose a critical threat. Thus, it is imperative that security mechanisms be developed to mitigate this threat.

[5] `register_globals` is a configuration flag that, when enabled, causes the PHP interpreter to automatically insert variables corresponding to various HTTP request parameters into the global namespace. This has the effect of potentially enabling an attacker to influence control flow or manipulate data through request parameter manipulation.

[6] `magic_quotes_gpc` is another configuration flag for PHP. It controls whether or not the PHP interpreter will automatically sanitize all input variables to a script in preparation for their use in a database query, and is intended to prevent SQL injection. Since a given script is independent of the configuration setting, however, one of two misconfigurations can occur. First, a script may assume that `magic_quotes_gpc` is enabled when it is not, in which case no protection against SQL injection is afforded. Second, a script may erroneously assume that `magic_quotes_gpc` is disabled. In this case, the script may simply cease to function correctly at all, since any input strings containing certain characters will be silently modified by the PHP interpreter.

1.2 Mitigating the threat

Strategies for dealing with security vulnerabilities can be broadly classified into four categories: avoidance, detection, prevention, and recovery. The following section introduces each approach and discusses their relative advantages and disadvantages.

1.2.1 Avoidance

The goal of avoidance is twofold. First, avoidance strategies attempt to prevent security vulnerabilities from being introduced into software during the development stage through the usage of secure coding and design techniques. Second, avoidance strategies preemptively identify and remove vulnerabilities from software before it is deployed in a security-critical situation.

Secure development practices

The area of secure development practices can be considered as a subset of the traditional field of software engineering, with a specific focus on security. As the field is broad, a full overview is elided; instead, a sampling of some well-known principles is provided.

One important theme of secure development practices is that of incorporating security planning into the development process from an early stage. This may take the form of specifying security policies to be enforced, or designing the architecture of the system to reduce the impact of vulnerabilities. Clearly, if security features are not carefully

considered during the design stage but are rather "bolted-on" after the fact, the likelihood that oversights or unforeseen consequences will occur is greater. Indeed, history is littered with spectacular security failures that can be directly attributed to the failure to apply this principle.

Another fundamental maxim of secure design is the principle of least privilege, which states that a subject should be given only those privileges necessary to correctly perform its task. Furthermore, if a subject only requires a limited set of privileges for certain tasks, the subject should either only acquire those privileges when necessary or it should be split into separate privileged and non-privileged subjects ("privilege separation").

A common implementation-level practice is to avoid the usage of APIs known to be conducive to introducing security vulnerabilities. This often arises when a function, or set of functions, is difficult or impossible to use in a safe manner. Perhaps the canonical example is that of the `strcpy()` function; since the programmer cannot supply an upper bound on the number of characters copied, it is only safe to use when the maximum length of the source buffer is known and can be verified to always be less in length than that of the destination. Since this is often not the case, this function has, in all likelihood, been responsible for more buffer overflow vulnerabilities than any other single cause in history.

A final example of secure design principles is that of avoiding "security through obscurity." The name originally refers to the practice of basing the security of a cryptographic algorithm around the secrecy of how the algorithm works, although the principle has since been generalized. This principle is generally interpreted as an observation that it is preferable to rely upon a well-designed security architecture rather than ignorance of

its insecurity.

Unfortunately, regardless of the mindfulness of software designers and developers, vulnerable software continues to be produced. Therefore, it is desirable to provide some level of quality assurance by identifying security vulnerabilities in software during or after development.

Identifying security vulnerabilities

In practice, identifying security vulnerabilities generally encompasses a plethora of techniques for the analysis of software, each of which may be classified along several axes: static or dynamic[7], white-box or black-box, and manual or automated.

Static analysis. Static analysis is performed offline on either source code or directly on an executable image. Because of the offline nature of static analysis, such techniques are by necessity considered white-box, where white-box refers to the ability of the technique to directly analyze the code or dynamic state of the software under test. This is as opposed to black-box approaches, which are restricted to providing inputs to the software and observing the external results; as the name suggests, these techniques approach the software under test as a "black-box."

On the other hand, static analysis may be considered as either manual or automated. Manual static analysis generally refers to the practice of code auditing, in which skilled

[7]Note that we define the terms static and dynamic with respect to vulnerability analysis in a wider sense than is generally used. In much of the literature, however, static and dynamic analysis are synonymous with static and dynamic code analysis.

security analysts examine the source code or executable image of the software under test in order to identify potential vulnerabilities. Code auditing has emerged as a particularly effective means of discovering software vulnerabilities, and is widely practiced by both industry and various government agencies. Nevertheless, the efficacy of manual techniques relies directly on the competency of the security analysts themselves, and the fallibility of these analysts as well as the increasing complexity of software that must be analyzed has prompted investigation into powerful automated techniques for discovering software vulnerabilities.

Automated static analysis operates by building a model of the execution of the software under test. This is typically performed by translating a representation of the software into an intermediate format suitable for a model-checker or SAT[8] solver such as SPIN [45], BLAST [42], SLAM [5], or MiniSAT [24], which then completes the analysis. Because it is generally prohibitive to consider every possible execution state of a program, some level of abstraction is induced to maintain the tractability of the analysis. This abstraction results in a sound and conservative analysis. The results are sound in the sense that they are accurate regardless of the input to the program. On the other hand, because of the abstracted nature of the model, the results are conservative in that they reflect weaker properties than may actually hold in practice. Despite this lack of precision, however, a major advantage of automated static analysis techniques is their potential to provide full coverage of all possible executions of a program.

Examples of well-known automated academic static analyzers include MOPS [13], MC [3], EXE/STP [12], and Saturn [136]. Additionally, despite the difficulties of mod-

[8]SAT is a shorthand reference to the Boolean satisfiability problem.

eling programs written in scripting languages popularly used for web application development, several static analyzers have been developed for PHP, including Pixy [54] and the tool presented in [137]. The viability of automated static analysis is further exemplified by the existence of commercial tools such as Coverity Prevent [18] and Fortify SCA [36], both of which are widely used.

Dynamic analysis. Dynamic analysis operates by observing the software under test as it executes. During execution, if an input causes or could potentially cause the program to enter a state that would violate a defined security policy, the analyzer reports the failure of the software to prevent itself from entering such a state as a vulnerability. Similar to static techniques, dynamic analysis approaches can be classified as either white-box or black-box. In the former case, the concrete execution states for the software under test with a given input are directly known, either by dynamically tracing the target program in a native environment or by leveraging a virtualized environment. In the latter case, a test driver supplies a variety of inputs to the software under test and observes the external results of the processing of these inputs. If the program exhibits behavior that is consistent with a security violation, a vulnerability is reported. Such test drivers are commonly known as "fuzzers."

Examples of both manual and automated dynamic analysis techniques exist. Manual dynamic analysis is more generally termed "penetration testing," in which teams of skilled security analysts attempt to "penetrate" the security defenses of a computer network or system.[9] In the case of software vulnerability analysis, this takes the form of demonstrating security vulnerabilities by attempting to bypass checks on program

[9] Alternatively, "red-teaming" or "tiger-teaming."

input that enforce a defined security policy.

Dynamic analysis, in contrast to static techniques, is considered precise in that no abstraction is introduced into the analysis. Instead, at a minimum, the input that caused the program to violate a defined security policy is directly known; in the case of white-box dynamic analysis, the exact set of program checks that allowed the program to enter the security-critical state are known. The major disadvantage, however, of dynamic analysis is its reliance on the quality of the set of inputs used. Inputs that are not representative of real-world usage of the software under test result in a lack of testing coverage of the software and, as a consequence, significantly degraded usefulness of the results. Regardless, dynamic analysis has gained in popularity due to the relative efficiency and precision of the approach.

Examples of automated dynamic analyzers include Nessus [119], Valgrind [111], Anubis [25], and Daikon [28].

The various avoidance techniques described above have proven effective at discovering software vulnerabilities, and are generally prescribed as elements of secure software development best practices. In particular, automated static and dynamic analysis techniques have made dramatic strides in the past decade, and continue to improve. Regardless, a common drawback to all avoidance approaches is that of completeness. Specifically, no analysis technique can guarantee the absence of security vulnerabilities in any piece of software. Fundamentally, this is a consequence of being reducible to the problem of program understanding, which is known to be NP-complete. Therefore, while avoidance of software vulnerabilities is an important component of an overall security strategy, it must be complemented with other approaches to mitigating software

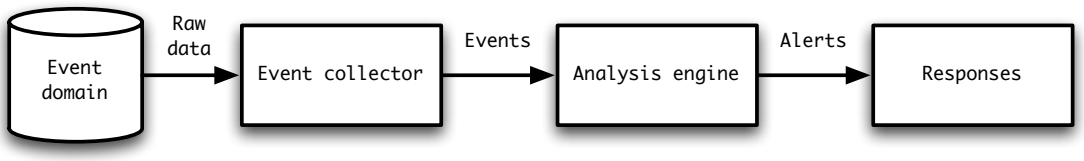


Figure 1.5: Generic architecture of an intrusion detection system.

security threats.

1.2.2 Detection

Detection mechanisms, as opposed to avoidance strategies, monitor deployed software in an attempt to identify when a system is under attack or has been compromised by an attacker. A system that employs such mechanisms is generally known as an intrusion detection system (IDS). Much work has centered around the design and evaluation of intrusion detection systems, and over time a standard architecture has emerged, to which almost all intrusion detection systems adhere.

Intrusion detection systems can generally be decomposed into three basic components, as depicted in Figure 1.5. First, an event collector component is responsible for monitoring and packaging events from a specific domain. This component may perform event normalization, feature extraction, or other preprocessing, but no analysis is usually performed at this level. Once these events have been prepared, they are passed to the analysis engine, the second component of an IDS. The analysis engine implements the actual detection methodology, which differs from system to system. However the analysis is performed, its fundamental purpose is to differentiate malicious traffic from

		Attack occured		
		p	n	**Total**
Attack detected	p'	True positive	False positive	P'
	n'	False negative	True negative	N'
	Total	P	N	

Table 1.4: Confusion matrix representing possible IDS decision outcomes.

normal traffic. Finally, intrusion detection systems contain a third component to execute responses to detected attacks. Responses may range from simply logging the occurrence of the attack to more complex actions, such as directly attempting to block the attack from succeeding or dynamically updating a separate firewall in order to block further occurrences of the attack. The types of responses an IDS can perform are, however, largely orthogonal to its main distinguishing characteristics: monitored domains and detection methodology.

Intrusion detection methodologies are primarily evaluated with respect to their ability to distinguish attacks from normal behavior. In particular, IDS designers attempt to both maximize the *true positive rate* and minimize the *false positive rate* of a proposed technique. A confusion matrix presenting the possible outcomes of an individual classification performed by an IDS is presented in Table 1.4. In this table, the events corresponding to normal and malicious behavior are represented by n and p, while classifications on the part of an IDS of the events as normal and malicious are represented by n' and p', respectively. The true positive rate (TPR) of an IDS is calculated as $\frac{pp'}{pp'+pn'}$, while the false positive rate (FPR) of an IDS is determined by $\frac{p'n}{p'n+n'n}$.

A popular method to evaluate intrusion detection systems is the Receiver Operating Characteristic (ROC) curve, an example of which is shown in Figure 1.6. ROC curves

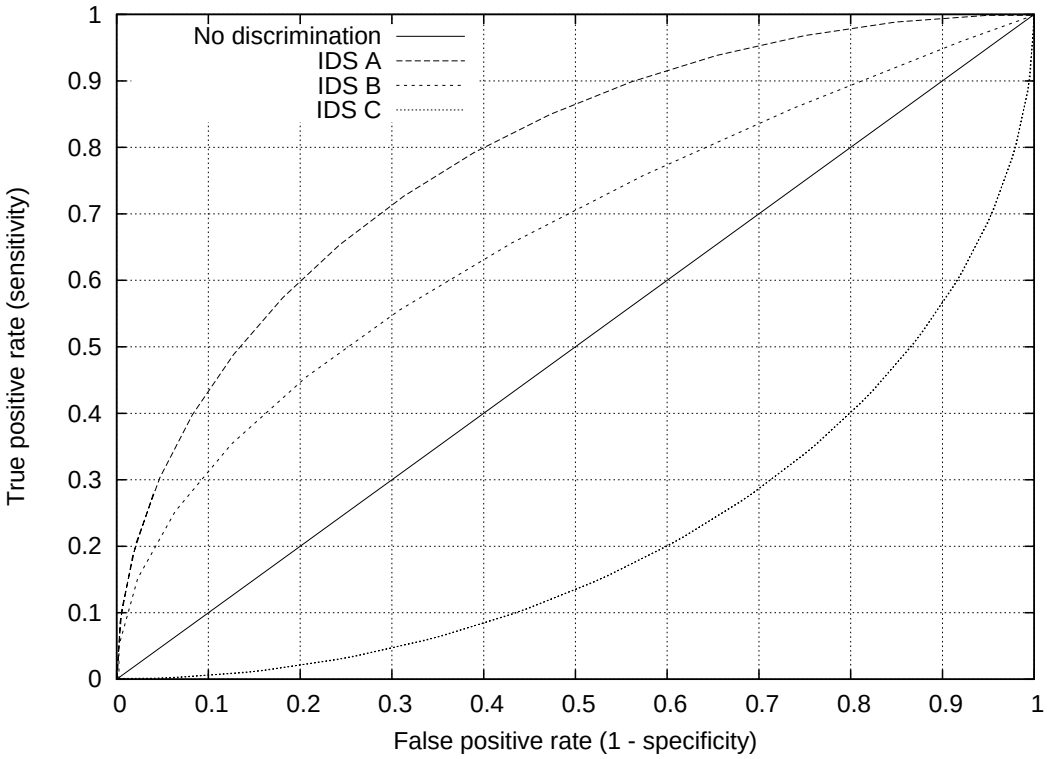


Figure 1.6: Example ROC curves.

were introduced during World War II in the field of signal detection for evaluating radar systems, and have since been applied to the evaluation of intrusion detection methodologies. ROC curves plot the true positive rate of an IDS as a function of its false positive rate, and are used to provide a graphical representation of the tradeoff associated with increasing the true positive rate, or sensitivity, of a detection methodology. $(0, 0)$ corresponds to classifying all events as normal, while $(1, 1)$ represents the classification of all events as malicious. Points along $f(x) = x$ represent a lack of discrimination, or a random guessing strategy. Points above this line represent positive discriminatory power, while points below the line represent negative discriminatory power.[10] IDS de-

[10]Note that such a system does in effect possess positive discriminatory power, achieved simply by

signers typically strive for plots approaching $(0, 1)$ – that is, the upper-left corner of the plot. A ROC curve for an individual IDS is used to determine an optimal operating point given a specified tolerance to false positives. Additionally, several systems may be compared on a single plot; for example, in Figure 1.6, IDS A exhibits higher sensitivity for a given false positive rate than IDS B.

Detection methodologies

There exist two main detection methodologies employed by intrusion detection systems: misuse detection and anomaly detection. Misuse detection systems contain a set of signatures, each of which describes a manifestation of an attack. A misuse-based IDS monitors streams of events and attempts to match a sequence of events to these signatures. If a match is observed, the IDS considers an attack to have occurred. In this sense, a misuse detection system implements a blacklist policy or negative model, where certain event sequences are considered attacks, and the rest are considered benign.

Anomaly detection systems, on the other hand, employ the dual strategy. Instead of modeling evidence of malicious behavior, anomaly-based systems model the normal behavior of events from the monitored domains. If a sequence of events deviates significantly from these models, the IDS considers those events to be evidence of an attack. Consequently, anomaly detection systems implement a whitelist policy, or positive model.

reversing each of its decisions.

Model construction

Intrusion detection systems can be further classified according to whether their models are specified or automatically generated. Misuse signatures are predominantly manually specified by skilled security analysts, usually in response to the proliferation of known exploits. In contrast, anomaly detection models have traditionally been automatically generated from training samples using algorithms borrowed from the machine learning community. This dichotomy does not always hold, however, since counterexamples for each methodology do exist. Recent systems that attempt to automatically generate misuse signatures for Internet worms have been proposed, and several anomaly-based systems have been developed that rely on manual specification of benign behavior.

Due to their blacklist policy as well as the precision with which misuse signatures can model malicious behavior, misuse-based systems typically exhibit a relatively low false positive rate when compared to anomaly detection systems. On the other hand, misuse-based systems are also notoriously bad at generalizing to new attacks; at best, such systems can, in some cases, detect variations on known attacks. In contrast, anomaly-based systems are able to detect previously unknown, or "0-day," attacks, as the detection methodology is independent of the precise manifestation of the attack. The main drawback of anomaly detection systems, however, is their relatively high false positive rate, which has hindered their wider deployment in real-world situations.

Monitoring domains

Finally, intrusion detection systems can be categorized according to the domains they monitor. Typically, these are some subset of the network, host, or application domains. Network-based intrusion detection systems monitor events at various levels of the network stack, from link layer frames to network streams. These systems have the advantage that a single instance can detect attacks against a large number of endpoints, although implementing more resource-intensive detection algorithms in high-bandwidth environments has proven problematic. Additionally, network-based systems are more prone to evasion attacks; due to the decoupled nature of their analysis from individual endpoints, models of the state of an endpoint can easily become desynchronized from the true state of the endpoint. Also, the increasing prevalence of network-level encryption poses problems of both scalability and management for network-based IDS.

Host-based intrusion detection systems instead monitor events generated by the operating system, which are predominantly system call executions or system library calls. Host-based approaches are generally harder to evade than network-based approaches, since a host-based IDS's view of the monitored system is much more tightly coupled. Regardless, evasion attacks against such systems have been demonstrated. In addition, host-based systems can incur a much higher deployment cost than network-based IDS, since instances must be installed, configured, and maintained on each individual endpoint.

Application-based intrusion detection systems monitor events from within individual applications running on an endpoint. These systems monitor application-specific events

or, alternatively, are built into the application itself. An application-based IDS is therefore extremely tightly coupled to the state of the application, and as such is potentially the most difficult to evade. Similar to host-based intrusion approaches, however, management costs are higher than in the case of network-based intrusion techniques. Also, concerns have been raised about the prospects for system performance degradation or denial of service due to the inline architecture of certain systems.

As intrusion detection is the main focus of this dissertation, a thorough discussion of individual intrusion detection systems will be deferred until Chapter 2.

1.2.3 Prevention

Prevention mechanisms share some degree of overlap with detection mechanisms in that they are both intended to identify the exploitation of a vulnerability. Once an attack has been identified, however, prevention mechanisms additionally endeavor to thwart the success of the attack. In this respect, prevention mechanisms can often be viewed as an intrusion detection mechanism that has been extended with a preventative response; the so-called intrusion prevention system (IPS) is an example of this phenomenon.

Such a view, however, is not necessarily correct. For instance, prevention mechanisms are often integral to the system itself. To illustrate, consider the StackGuard buffer overflow prevention system [19]. Due to the integration of the system into the C compiler, it is impossible to perform stack-based buffer overflow exploits. This prevention mechanism therefore is an explicit consequence of the design of the system.

Prevention mechanisms are also often confused with avoidance mechanisms. Here, the distinguishing characteristic is when attacks are prevented. Avoidance mechanisms prevent the exploitation of vulnerabilities before they occur, while prevention mechanisms block attacks in an online manner. To illustrate, consider the Java programming language and virtual machine runtime. Due to the design of the language and bytecode verifier, it is impossible to perform arbitrary memory overwrites; this security property is verified before code is allowed to execute. In contrast, StackGuard prevents stack overflows at runtime.

Another distinguishing characteristic is that, as also illustrated in the previous example, prevention mechanisms are often targeted towards a specific class of vulnerability. To wit, consider the succession of prevention mechanisms intended to prevent stack-based or heap-based buffer overflows. Each of these systems does nothing to address, for instance, format string vulnerabilities, function pointer overwrites, or brainwashing attacks, and furthermore they cannot do so without extending the system itself. This specificity of design and intent is indicative of prevention mechanisms.

Prevention mechanisms generally possess the advantage that, due to their narrow focus, they are highly effective in thwarting a single class of vulnerabilities. Nonetheless, prevention mechanisms do suffer from several drawbacks. In particular, prevention mechanisms generally require a well-defined security policy to enforce. For instance, a succinct policy for a stack-based buffer overflow prevention system might be "never allow a user-initiated memory write to overwrite a system pointer." For a system login program, the policy might be "never allow a user to access the system without providing the correct password." These policies are simple, distinct specifications, and

consequently their enforcement is straightforward. As security policies become more complex or nebulous in their specification, however, it becomes accordingly more difficult or, in some cases, impossible to efficiently and effectively enforce the policies.

Another disadvantage of prevention mechanisms is that they can be cumbersome to deploy and, as a result, are less effective than they would otherwise be. Stack-based buffer overflow prevention systems, for example, are well-known for causing seemingly arbitrary program crashes due to previously unknown buffer overflows in those programs. In many of these situations, instead of auditing the source code to remove the bug, these systems were simply turned off. A similar phenomenon was observed for systems intended to prevent the execution of code on the stack or heap. Certain programs, such as just-in-time (JIT) compilers, legitimately require the ability to execute outside of the code segment. For such systems, as in the previous case, non-exec prevention mechanisms had to be disabled.

Finally, prevention mechanisms themselves can be exploited by an attacker to cause a denial of service. This follows from the observation that the actions performed by a prevention mechanism in response to a detected attack are generally severe. For instance, a buffer overflow protection system may be forced to terminate a process in which an overflow has occurred, since critical data controlling application behavior and control flow might have been corrupted. Similarly, an IPS may insert blocking rules into a firewall in response to a detected attack. If an attacker is able to spoof their identity such that the ensuing response is targeted towards others, a denial of service may be created. The potential for this situation to arise has had real consequences. For instance, many businesses have been reticent to adopt intrusion prevention systems due

to the loss of revenue associated with service downtime.

Examples of prevention mechanisms abound. There have been many systems to prevent stack-based buffer overflows, including StackGuard [19] and SSP [30]. Similar systems have been proposed to protect heap data structures [104]. Both OpenBSD [95] and PaX [96] incorporate a number of prevention mechanisms; examples such as non-executable page protections and address space layout randomization have proven influential. Prevention mechanisms also exist for mitigating XSS [127] and CSRF [53] vulnerabilities.

1.2.4 Recovery

Recovery strategies approach the problem of threat mitigation from a different perspective than that of prevention strategies. Rather than attempt to thwart the success of an attack, recovery mechanisms allow systems to continue operating in the face of successful exploitation. There are two main classes of recovery systems: those that discontinue service while recovery is performed, and those that allow service to continue during recovery, albeit in a potentially degraded capacity.

Recovery mechanisms that result in an interruption in service overlap considerably with the traditional mechanisms of disaster recovery. The canonical example is the quintessential system administration task of creating regular backups of critical data. In the event of the corruption or destruction of data, a known good snapshot can be restored within a short time period. Because recovery generally involves identifying and removing the vulnerabilities that allowed the system to be compromised in the

first place, offline recovery mechanisms can also include forensic analysis of the target system.

Online recovery mechanisms generally borrow ideas from the area of fault tolerance, and are less commonplace than offline methods due to several factors. First, recovery mechanisms are usually inextricably bound to a given system or domain to be protected, since the act of recovery requires an intimate knowledge of the internals of the protected resource. Second, in many cases it is difficult to guarantee that the recovery will always succeed. Nevertheless, research continues to grapple with online recovery. Examples include the selective rollback of malicious database transactions [2], and recording snapshots of program states that can be restored in the event of an otherwise fatal system error [112].

A crucial disadvantage of recovery mechanisms is that they do not protect against unauthorized data disclosure; once data has been exposed, confidentiality cannot be restored. In the context of web applications, confidentiality of personal records is the overriding concern, and as a consequence recovery mechanisms are an ineffective means of threat mitigation.

1.3 Anomaly-based intrusion detection for web applications

Each of the mitigation strategies discussed in Section 1.2 has both advantages and disadvantages as they apply to the context of web security. In accordance with the security

principle of defense in depth, it is desirable that mechanisms embodying each approach be deployed to counter the threats to web applications. Nevertheless, in this section we will examine the suitability of each approach with respect to mitigating threats to web applications, and justify the approach we have adopted for this dissertation.

Avoidance mechanisms are clearly advantageous in that they offer the potential to identify the presence of vulnerabilities before they can be exploited. The impossibility of doing so with absolute certainty, however, forces one to conclude that other strategies must be employed in order to address those vulnerabilities that will inevitably remain undiscovered.

Prevention mechanisms, then, would seem to be an ideal strategy to compose with that of avoidance. The inclusion of systems to prevent, with a high degree of confidence, the exploitation of known web application vulnerabilities would theoretically provide maximal coverage against current threats. Several problems remain, however. First, any such protection mechanisms would only be guaranteed to be effective against known attacks; novel classes of vulnerabilities would not necessarily be addressed. Furthermore, protection mechanisms typically incur a high deployment cost that may prove unfeasible in practice. A protection mechanism that required, for instance, the modification of all web clients and servers in order to guarantee protection would be likely to face considerable obstacles in terms of adoption.

Recovery mechanisms may prove useful in terms of service availability and data integrity. Unfortunately, since many attacks against web applications are motivated by the desire to gain illicit access to sensitive data, recovery mechanisms would seem ill-suited to addressing threats to data confidentiality.

Detection mechanisms, therefore, are likely to offer the most effective mitigation strategy against both known and unknown threats to web applications. In this regard, anomaly-based intrusion detection systems are the most promising class of detection mechanisms. Furthermore, the relatively low deployment cost of intrusion detection, when compared to other strategies, increases the likelihood that effective detection systems will be adopted in practice.

Though both industry and government have recognized the need for advanced web application intrusion detection systems, many open challenges remain to be solved before such systems can be effectively deployed in the real world. First, previous anomaly detection systems have exhibited unacceptably high false positive rates, limiting the effectiveness of their reporting. Anomaly detection systems are also highly dependent on the quality of the training data. Noisy training data that inadvertently contain attack samples can render an anomaly detection system useless. Additionally, a lack of insufficient training samples to build accurate models can also result in a degradation of the ability of an anomaly detection system to correctly recognize attacks. Anomaly detection systems are also currently unable to recognize when the underlying web application has changed; in the machine learning community, this is referred to as the problem of concept drift. Accordingly, they are unable to distinguish between anomalies due to attacks and those due to inaccurate models. Anomaly detection algorithms historically have also exhibited high overhead, preventing their deployment in high-bandwidth environments. Finally, anomaly detection systems exhibit poor explanatory power regarding the attacks they detect. While misuse-based systems, due to their precision, are able to report the exact type of attack detected with a high degree of certainty, anomaly-based systems have heretofore been restricted to reporting that a statistically

significant deviation from a model has been identified.

In this dissertation, I make the following contributions to the area of web security.

- I present the design and implementation of WEBANOMALY, an advanced black-box anomaly detection system that accurately detects attacks against web applications with low performance overhead.
- I propose and evaluate *anomaly signatures*, a novel method for clustering anomalies in order to reduce the effective false positive rate of anomaly detection systems.
- I apply attack classification techniques to clustered anomalies in order to improve the explanatory power of anomaly detection systems.
- I introduce the problem of incomplete training data for web application anomaly detection systems due to the frequent non-uniformity of resource invocations. I propose and evaluate an effective technique to compensate for a local scarcity of training data by exploiting global similarities in web application anomaly models.
- I introduce the problem of web application concept drift, or legitimate changes in the behavior of web applications over time. I propose and evaluate the use of response modeling techniques to distinguish between anomalies caused by legitimate changes and those caused by malicious behavior, allowing an anomaly detection system to selectively re-train models without being vulnerable to model poisoning attacks.

- I present the design and implementation of a web application development framework that treats both web documents and database queries as strongly typed algebraic data types instead of unstructured sequences of bytes, automatically enforcing a separation between the structure and content of these objects. Due to the typing system imposed by the framework, applications are secure against major classes of attacks by construction.

The remainder of this dissertation is organized as follows. Chapter 2 presents an overview of related work in the field of intrusion detection and web security. Chapter 3 introduces the design of WEBANOMALY. Chapter 4 dicusses a technique to reduce the effective false positive rate of anomaly detectors, and a technique to increase their explanatory power. The problem of training data scarcity and how WEBANOMALY addresses this issue is discussed in Chapter 5. In Chapter 6, techniques for detecting legitimate changes in web applications and updating models to reflect those changes are presented. Chapter 7 presents a web application development framework that results in web applications that are secure by construction against popular classes of attacks. Finally, Chapter 8 draws conclusions and proposes areas for future work.

Chapter 2

Related Work

The field of intrusion detection is broad, and encompasses a substantial body of research into the design and evaluation of effective intrusion detection methodologies. The majority of this chapter will discuss significant contributions to the field, with the principal intent being to place the system described in this dissertation in the context of past work.

The standard taxonomy for intrusion detection systems involves classifying each system according to several distinguishing characteristics: detection methodology, either misuse-based or anomaly based; detection domain, either the network, the host, application, or some combination; and whether detection models are manually specified or automatically generated. We begin by discussing the foundations of research into intrusion detection.

2.1 Foundations of intrusion detection

The modern understanding of intrusion detection was introduced in a seminal paper by Denning [22]. The author proposes a general framework for detecting attacks against computer systems by modeling normal behavior patterns generated by users of the system. The implementation of this framework, called IDES, accomplishes this by analyzing system audit records generated by user logins, program executions, or file and device accesses. From these audit records, three metrics are derived: an event counter, which records the number of events of a certain type that occur during a given time interval; an interval timer, which records the length of time between two related events; and a resource measure, which records the utilization of a given resource for each event. Once these metrics have been calculated, a number of statistical models are generated in order to characterize the normal behavior of these metrics. These models include the operational model, which compares a metric against fixed thresholds; the mean and standard deviation model, which uses Chebyshev's inequality to establish loose bounds on the variance in the mean and standard deviation of a metric; the multivariate model, which, as its name suggests, extends the previous model to multiple random variables; the Markov process model, which treats an event counter as a state variable and characterizes the probability of the metric transitioning from one state to another; and the time series model, which characterizes the order and intervals between event counter or resource measures. These models are then composed in various ways into profiles, which are used to characterize the normality of different aspects of system behavior. Therefore, IDES implements what would today be termed a host-based anomaly de-

tection system.[1] Subsequent systems extended the anomaly-based model to include a rule-based component; for examples, see [79] or [49].

In [41], Helman and Liepins present a theoretical foundation for anomaly-based intrusion detection. The authors model event streams as a pair of stationary stochastic processes: the legitimate process N, and the malicious process M. Using this characterization, the authors provide formal bounds on the effectiveness of intrusion detection systems as a function of the difference of the densities of N and M over the space of all possible events. With perfect information about both N and M, the bound is shown to be achievable. Due, however, to the more realistic scenario where only imperfect knowledge of either process exists, the authors show that detection accuracy is typically far from this bound.

One of the first systems to propose misuse-based techniques as the sole means of detection was STAT [47]. STAT was designed to perform stateful analysis of event streams, modeling multi-step attacks as a directed graph of states and transitions collectively called attack scenarios. Scenario states are considered abstract representations of the state of the monitored system, and scenario transitions represent possible changes in the state of the monitored system. Each scenario has a single initial state and a set of final states, each of which corresponds to a successful attack. At system initialization, a single scenario instance, or unique instantiation of the scenario, is created at the initial state. Each scenario instance contains information about an attack in progress. As events are input into the system, the analysis engine attempts to match outgoing transitions at any state associated with a scenario instance. If the event type and an optional

[1] Interestingly, the author also anticipates the misuse-based detection methodology, and dismisses the idea as "too complex" and "unable to cope with intrusions that exploit deficiencies that are not expected."

transition assertion both match, the transition is taken. Depending on the type of transition, the scenario instance may move completely to the new state; the scenario instance may clone itself, with one instance remaining at the current state and the other moving to the new state; or, the scenario instance may destroy itself. Each scenario state or transition may also have an associated set of actions that are executed when a scenario instance enters a state or follows a transition; these are typically used to record relevant information about an attack, or to execute a response to an attack. The original work presented and evaluated USTAT, a system for monitoring Solaris Basic Security Module (BSM) audit records. Later work, however, generalized the system, abstracting the attack specification language and analysis engine from domain-specific event providers and custom response modules. The resulting framework was successfully applied to the network domain [123] and application domain [125], among others.

2.2 Misuse-based detection

In [69], Kumar and Spafford describe a system for misuse-based detection that bears a striking resemblance to STAT. In this work, the authors propose the use of Colored Petri Nets (CPN) to model multi-step attacks. In this system, each signature is represented by a CPN that has one or more start states, a single final state corresponding to a successful attack, and a set of transitions between these states, each of which may have an optional guard assertion. Current stages of an attack are represented by tokens, each of which contain a set of variables encoding information about the attack. If a token reaches the final state of a CPN, the corresponding attack is considered to have been matched. As

opposed to the previous work, however, matching is relatively expensive; in fact, it is exponential in the size of the input. Furthermore, no evaluation of the system has been presented to date.

Lindqvist and Porras summarize the design of the misuse-based detection component of the various SRI intrusion detection systems in [75]. The system, called P-BEST, is a forward-chaining expert system that encodes signatures as a set of facts. As events are processed by the system, an inference engine attempts to match events to existing facts in order to derive new facts through successive applications of the logical rule of *modus ponens*.[2] If an observed sequence of events causes a fact corresponding to a successful attack to be derived, an alert is generated. A custom language for describing fact-based signatures can be compiled into C language source code, from which either stand-alone executables or libraries can be produced. The system itself is independent of the detection domain, having been applied to BSM audit logs as well as network traffic.

Snort [107] is a network-based IDS that has enjoyed considerable popularity due to its relative simplicity and open-source development model. Snort signatures are written in a custom rule language targeted entirely at expressing constraints on network packet headers and payloads. In its most basic form, Snort is stateless in that it considers each network packet independently. Limited support for stateful analysis exists, however, through a number of preprocessors for reassembling network streams or detecting portscans. Additionally, a simple mechanism for chaining rules exists.

In [97], Paxson describes the Bro network-based IDS. Bro was designed to perform

[2] *Modus ponens* states that $(p \wedge (p \rightarrow q)) \rightarrow q$.

real-time intrusion detection for high-speed links, and was written to resist attacks against the system itself, albeit under the assumption that attacks will only originate from outside a trusted enclave. The system utilizes a custom rule language to express its attack signatures, and furthermore is extensible to allow for service-specific analysis. Bro was successfully deployed at LBNL, and has since become more widely used as an alternative to Snort.

Almgren and Lindqvist propose in [1] one of the first application-based intrusion detection systems, where the detection system is integrated into the monitored application itself. The system described was implemented as a module for the Apache web server, though the approach is independent of the application. This technique has several advantages over more traditional network-based or host-based intrusion detection. Crucially, desynchronization attacks against the monitored application are essentially impossible, as the IDS has a complete view of the state of the application as it processes events. Also, the IDS is able to inspect events that would otherwise be encrypted, since it has access to the data in unencrypted form. An additional benefit is that the IDS need not concern itself with events unrelated to the monitored application, as in the case of network-based IDS. Finally, attack response mechanisms can take advantage of existing error handlers within the application itself instead of more traditional and inelegant methods such as connection termination or blocking.

Application-based intrusion detection does have some drawbacks, however. Due to the tight integration, detection overhead may impact the performance of the monitored application. Also, application-based IDS development does not scale as well as other approaches, since each application to be monitored requires some development effort

to integrate the IDS. Additionally, a successful attack against the monitored application may allow for the possibility for the attacker to disable the IDS. Finally, deployment costs are higher due to the overhead of managing a number of detection systems as opposed to, for instance, one NIDS. Regardless, this approach has proven viable; the system described in [1] was later reimplemented under the guise of a web application firewall as mod_security [102].

2.3 Anomaly-based detection

Forrest *et al*. frame the problem of intrusion detection as as an immune system that distinguishes between the self, or normal behavior, and the other, or malicious activity [35]. The authors propose the automated learning of sequences of system calls to accomplish this; therefore, the system described is, at heart, a host-based anomaly detection system. During the training phase, sequences of 5, 6, and 11 system call invocations are observed from executing processes. Only the type of the system call is recorded, while the arguments, return value, and other information is discarded. During the detection phase, a number of deviations above a certain threshold from the established database of system call sequences observed during the training phase are reported as attacks. The metric used in this work was the Hamming distance, although others are possible. This approach was later extended in [44], and prompted further research into system call-based anomaly detection.

In [133], Warrender *et al*. explore alternate approaches to analyzing sequences of system calls. In this paper, they compare several methods of analysis, including sequences

of system calls, relative frequencies of system call sequences, rule induction, and Hidden Markov Models (HMM). Their analysis concludes that, although HMMs exhibited the best detection accuracy, their high computational cost outweighed their benefit in light of the comparable accuracy of more lightweight methods.

Wagner and Dean describe a system for learning normal sequences of system calls statically from application source code in [129]. The system they propose uses static analysis techniques to build a profile of legitimate system call sequences for each application to be protected. Three methods for building such profiles are described: call graphs, which can be represented as non-deterministic finite automatons; abstract stack modeling, the output of which can be represented as a non-deterministic pushdown automaton (NDPDA); and digraphs of k-sequences of system calls. Both system call types as well as those system call arguments that can be statically determined are considered. The resulting system demonstrated promising accuracy, but the approach suffers from the drawback that static analysis cannot model application behavior with sufficient precision to avoid the possibility of generating false positives. Also, both the NDFA and NDPDA models exhibited exponential running time in several cases. Finally, this paper is notable for introducing the mimicry attack against anomaly detection systems, where an attacker can evade detection by mimicking a legitimate sequence of system calls in the course of the attack.

Sekar *et al*. [110] apply the automaton construction approach introduced by [129] to a dynamic context. In this work, sequences of system call traces are observed by tracing an application from another user-level process. System call types themselves are discovered by analyzing the stack of the monitored application. The resulting automaton

is then used during the detection phase to identify deviations, which are reported as attacks. Because the technique is dynamic, the resulting automaton is precise, because the state of the monitored application is known. On the other hand, the technique suffers from the usual drawback that the coverage of the learned automaton is dependent on the inputs to the monitored application during the learning phase.

Mutz *et al*. propose two improvements to system call-based anomaly detection in [88]. First, in addition to modeling normal sequences of system calls, the arguments to these system calls are also considered. The use of system call arguments to enhance the discriminatory power of system call anomaly detection is motivated by the observation that advanced attacks against applications may only manifest themselves in argument values. Additionally, the authors evaluate the use of Bayesian networks to capture conditional dependencies between individual model score outputs that are not well-modeled by simple weighted sum score combinations. The resulting system demonstrated significant improvements in detection capability over previous systems for certain classes of attacks.

In [62], Ko *et al*. describe what they term a specification-based approach to intrusion detection. In their system, legitimate sequences of execution events are manually specified using a specialized grammar. This grammar is capable of capturing temporal security properties that may exist in a parallel or distributed environment. Deviations from these specifications of normal behavior are considered to be evidence of malicious behavior. In the context of previous work, the specification-based approach can be considered an example of anomaly-based intrusion detection where the models are manually specified. Indeed, Ko later applied machine learning techniques, specifically Inductive

Logic Programming (ILP), to derive models of normal program behavior [61].

Ghosh *et al*. propose the use of neural networks for intrusion detection in [38]. During the training phase, a combination of external inputs and internal state of monitored applications is provided to a backpropagation network. During the detection phase, the resulting neural network was used to classify input events as either normal or anomalous. The system described, however, was only evaluated on a single application with synthetic training data. Also, concerns over the scalability of this approach have been raised, as backpropagation networks are expensive to train.

The use of Instance-Based Learning (IBL) techniques to perform anomaly detection is proposed by Lane and Brodley in [71]. In this work, feature vectors are extracted from UNIX shell commands and compared to historical profiles using a similarity measure. A sequence of real-valued similarity measures is then processed by a noise-suppression filter. The smoothed data stream is then classified by a threshold decision module, where the decision boundaries have been specified by manual analysis of historical user behavior. The proposed system also includes a feedback loop to update user profile and classification parameters; this is intended to address the problem of concept drift. Additionally, two approaches to reducing data storage requirements for the system were evaluated: LRU pruning, and clustering using both K-centers and greedy algorithms. The system was evaluated over training data accumulated from real UNIX systems, and was shown to produce results with good accuracy. A drawback of the system, however, is its offline nature, which impacts both the currency of its alerts as well as, in some cases, its accuracy.

Valdes and Skinner describe the use of Bayesian networks for network-based anomaly

detection in [122]. In particular, the authors propose the analysis of TCP stream burstiness to discriminate between normal and anomalous traffic. The Bayesian network they describe contains evidence nodes for various features of network traffic, and a single hypothesis node that can assume belief values over a range of normal and anomalous values. A monitor updates the evidence nodes with observations from the network, and a Bayesian inference engine classifies the traffic at the hypothesis node using well-known belief propagation algorithms. The authors also describe means by which the network can adapt to concept drift, either by updating the conditional probability tables corresponding to each evidence node, or by dynamically adding new hypotheses to the root node. Although the system performed well over the 1999 MIT Lincoln Labs IDEVAL data sets, it fared less well on real-world data. Also, the proposed system has been criticized for its use of a naive Bayesian network. This criticism is mainly due to the fact that naive Bayesian networks fail to model conditional dependencies between evidence nodes, and devolve in practice to the weighted sum approach.

In [72], Lee and Stolfo propose a general framework for anomaly detection, called MADAM ID, that incorporates data mining techniques to improve detection capability. In contrast to other machine learning-based anomaly detection systems, where features are chosen manually by domain experts, MADAM ID also uses data mining algorithms to automatically identify the features to extract. First, association rules, or multifeature relationships, are automatically derived for both the training set of normal data and an attack training set. These rules are then filtered by domain experts to include only those features essential to facilitating detection. Then, the rules generated by processing both data sets are compared to identify those rules that accurately discriminate between normal and malicious behavior. The system was evaluated against the 1998 MIT Lincoln

Labs IDEVAL data set, and ranked as one of the most effective systems tested. A significant caveat, however, is that the system requires a labelled attack data set to identify discriminating rules.

Lee and Xiang propose the use of information theoretic measures to provide a framework within which anomaly-based intrusion detection systems can systematically be improved in [73]. In particular, the authors observe that lower entropy data is likely to produce simpler, more robust models due to its inherent regularity. Similarly, the authors describe how conditional entropy can indicate whether sequences of features are deterministic. Relative entropy, or the Kullback-Leibler divergence, is proposed as a measure of the similarity of two data sets. Finally, information gain is proposed as a measure of the discriminatory power of event features. The authors also present several case studies demonstrating how these measures may be used in practice.

One of the first attempts to apply anomaly detection techniques to application-level data is due to Kruegel et al. [67]. The system described is network-based, and is composed of two high-level modules: the Packet Processing Unit (PPU) and Statistical Processing Unit (SPU). The PPU is responsible for normalizing network streams and performing service-specific protocol parsing. The actual detection methodology is implemented by the SPU. Protocol messages are first grouped according to type; for instance, HTTP GET requests are considered separately from HTTP POST requests. Then, an anomaly score for each message is obtained by computing the weighted sum of various statistical measures applied to three request features: type, length, and payload distribution. The system is evaluated against data sets of HTTP and DNS traffic, and exhibited promising detection accuracy. Indeed, the approach proposed by this paper heralded much of the

later research into web application anomaly detection.

Kruegel *et al*. present a multi-model approach to web application anomaly detection in [68]. In this paper, the intrusion detection system processes web server access logs, from which statistical models are learned for each unique request path and parameter. These models include the token model, which records the set of unique values a particular parameter can take; the length model, which uses the Chebyshev inequality to derive loose bounds on the length of a parameter value; the character distribution model, which constructs an idealized distribution of the characters comprising observed parameter values; and a structural inference model, which uses established HMM induction techniques to build a probabilistic grammar that captures normal parameter structure. The system was evaluated over real-world data sets, and was shown to exhibit high detection accuracy and a low false positive rate. Additionally, the multi-model approach was validated by demonstrating that individual attack types were better detected by different subsets of the individual models.

Wang and Stolfo describe a service-agnostic network-based anomaly detection system called PAYL in [132]. The PAYL system characterizes normal behavior by modeling byte frequency distributions of network packet payloads. During the training phase, distributions are computed for each packet observed; separate distributions are maintained for each network port, stream direction, and payload length. At the end of the training phase, a clustering algorithm is applied to merge similar distributions for a given port and stream direction. The clustering algorithm uses the Manhattan distance to repeatedly compare distributions of consecutive length, merging similar distributions until the distance between each resulting cluster is above a certain threshold. Then, during the

detection phase, byte frequency distributions for observed network payloads are compared against the learned models using a simplified Mahalanobis distance. Significant deviations from the models result in an alert. The authors also propose the Z-string, which is simply a ranked byte frequency distribution, as a canonical form of learned models that can be used as a type of signature.[3] The system was evaluated against the 1999 Lincoln Labs IDEVAL data set.

The PAYL system was further extended in [130] by Wang, Cretu, and Stolfo to generate multiple centroids per observed payload length as well as to correlate ingress-egress alerts in order to detect worm propagation. In particular, the authors evaluate the use of string equality, longest common substring, and longest common subsequence matching to measure the similarity between suspected worm payloads. PAYL, however, was later demonstrated to be vulnerable to mimicry attacks due to blended, polymorphic worms by Kolesnikov et al. [63]

In response, Wang *et al*. proposed the use of higher-order n-grams in [131]. Due to the exponential growth in memory overhead and required training set size as n increases, the authors utilize Bloom filters to record n-grams observed from packet payloads during the training phase. This has the benefit of low computational overhead, as well as removing the restriction that n be fixed. During the detection phase, packet payloads are scored according to the proportion of n-grams observed that are not contained in the Bloom filter. The authors further propose a criterion for automatically determining when a model has been sufficiently trained by computing the likelihood of observing new n-grams. Finally, the authors describe a process of randomized modeling, where

[3] Incidentally, the Z-string exhibits striking similarity to the idealized character distribution described in [68].

multiple models are computed over a fixed, secret partition of packet payloads. The partitioning is considered equivalent to a secret key that cannot be guessed by attackers attempting to thwart the n-gram analysis through the use of exploit fragmentation and payload padding.

Ingham *et al*. propose a DFA induction approach to modeling normal HTTP requests in [48]. During the training phase, HTTP requests are normalized and tokenized according to a set of heuristics. The Burge DFA induction algorithm is then applied to the observed token sequences, and a separate DFA compression algorithm is applied at regular intervals. The compression algorithm has the effect of introducing a limited amount of generalization into the resulting DFA. Then, during the detection phase, observed token sequences are compared to the learned model by determining whether the induced DFA could potentially generate the observed sequence. The comparison algorithm, however, tolerates underivable tokens by recording the event and attempting to resynchronize with the DFA. A similarity measure between the observed token sequence and the DFA is then calculated as the proportion of tokens reached by valid transitions over the total number of tokens in the sequence. Additionally, when a token sequence exhibits a strong, but imperfect, similarity to the model, the DFA is modified to incorporate new states and transitions to reflect the observed token sequence. This process introduces a nonstationarity property that is intended to reduce false positives and address the issue of concept drift. Unfortunately, the resulting system exhibits a significant false positive rate, and the incremental updating performed on the DFA during the detection phase renders the system vulnerable to attacks that slowly poison the DFA to accept malicious requests as normal.

A recent effort on addressing training set deficiencies has been proposed in [20]. In this work, a sanitization phase is first performed to remove suspected attacks and other abnormalities from the data. Instead of creating one model instance, a set of "micro-models" is trained against disjoint subsets of the training data. These micro-models are then subject to one of several voting schemes to recognize and cull outliers that may represent attacks.

In [37], the authors propose a cluster-based anomaly detection system as a means of reducing false positives. The system accomplishes this by clustering similar behavioral profiles for individual hosts using the k-means algorithm, although the exact distance metric used was not explicitly given. Then, alerts are generated according to a voting scheme, where the causal event for an alert is evaluated against behavior profiles from other members of that host's cluster. If the event is deemed anomalous by all members of the cluster, an alert is generated.

In [82], a mixture of machine learning techniques is exploited to detect anomalous system calls in the Linux kernel. Ad-hoc distances between system calls are defined to perform clustering in order to identify natural classes of similar calls. The reduced size of the clustered input makes the training of Markov chains efficient. The behavior of each host application is modeled as Markov chains on which probabilistic thresholds are calculated to detect misbehaving sequences.

A recent proposal for the anomaly-based detection of web-based attacks is presented in [113]. In this work, a mixture of Markov chains incorporating n-gram transitions is used to model the normal behavior of HTTP request parameters. The resulting system attains a high detection accuracy for a variety of web-based attacks.

HTTP responses are exploited in [141]. In addition to other features, the DOM is modeled to enhance the detection capabilities of SQL injection and cross-site scripting attacks.

2.4 Evaluating intrusion detection systems

In [100], Puketza *et al*. propose a general methodology for testing intrusion detection systems. To guide their testing approach, the authors identify several broad performance objectives for intrusion detection systems. First, the IDS should be able to accurately identify intrusions. Second, the IDS should maintain its resource consumption within reasonable limits. Finally, the IDS should be resilient to "stressful" conditions, such as attacks against the system itself. The authors then propose the use of the `expect` tool as part of a framework to automatically execute test cases against an IDS under test. The testing framework is capable of generating both normal background traffic as well as attacks, and includes a means of synchronizing test cases to facilitate deterministic testing. The framework is then used to demonstrate how an IDS might be evaluated according to the performance objectives previously identified.

Axelsson highlights an important, and previously neglected, consideration when designing and evaluating intrusion detection systems in [4]. Specifically, the author applies the well-known base-rate fallacy to the problem of intrusion detection. The base-rate fallacy itself is a direct consequence of Bayes' theorem, and states that an interpretation of the conditional probability of an event given evidence of that event must take into consideration the prior probability, or base-rate, of that event actually occurring.

In the context of intrusion detection, the author shows that since the prior probability of an attack occurring in most environments is generally low, intrusion detection systems must attain extremely low false positive rates in order to achieve an acceptable detection rate. The author concludes that the proper optimization strategy for designers of intrusion detection systems is not to maximize the detection rate, but rather to minimize the false positive rate. Consequently, minimizing the false positive rate became a central design goal of subsequent intrusion detection systems.

Lippmann *et al*. report on a milestone event in the evaluation of intrusion detection systems in [76]. Sponsored by DARPA, the 1998 MIT Lincoln Labs Offline Intrusion Detection Evaluation was the first organized large-scale attempt to evaluate the state-of-the-art in intrusion detection. In the course of the evaluation, background traffic for a network of thousands of hosts with hundreds of users was simulated. 300 instances of 38 distinct attacks of various types were also introduced, and a number of intrusion detection systems were evaluated on the resulting data sets. As a result of this evaluation, the authors concluded that the ability of the then-current generation of intrusion detection systems to detect novel attacks was unsatisfactory. Due to the paucity of high-quality training data, the 1998 and 1999 MIT Lincoln Labs IDEVAL data sets became very influential in the evaluation of subsequent intrusion detection systems.

The MIT Lincoln Labs IDEVAL data sets were not without their detractors, however. One published critique is due to McHugh [84]. While the author is careful to emphasize the point that the IDEVAL data was valuable in that no other data comparable in quality was available to IDS researchers, he nevertheless points out several shortcomings. First, the author criticizes the noticeable absence of commercial systems from the evaluation,

which he claims is due to an assumed inferiority of such systems on the part of the evaluation organizers. Second, the author attacks the means by which background data was generated, highlighting the absence of any quantifiable measure of the similarity of the background traffic to real samples. Third, the realism of the attack traffic is called into question, particularly the distribution of attack instances among the background traffic. Similar critiques are levied against the simulated network architecture. As a final example, the taxonomy of attacks is attacked for several reasons. Due in part to McHugh, awareness of the difficulties involved in evaluating an IDS against synthetic traffic slowly permeated through the research community. This prompted researchers to, at a minimum, avoid relying solely on the IDEVAL data sets to evaluate proposed intrusion detection systems.

2.5 Attacking intrusion detection systems

The first substantial work documenting ways in which intrusion detection systems could be attacked is due to Ptacek and Newsham [99]. In particular, the authors identify two main problems with misuse-based network intrusion detection systems. First, network-based IDSs are susceptible to desynchronization attacks, in which the modeled state of network traffic and endpoint state is forced by an attacker to diverge from their true state. Second, due to the requirement that they must protect multiple, perhaps many, endpoints, network-based IDSs are vulnerable to denial of service attacks. The authors identify three classes of desynchronization attacks that can be mounted: insertion, evasion, and ambiguities. Insertion attacks involve identifying network packets that an

IDS will accept as legitimate, but an endpoint will discard. Evasion attacks refer to the transmission of packets that an IDS will reject, but an endpoint will accept. Finally, ambiguity attacks exploit a lack of knowledge on the part of the IDS concerning corner case endpoint behavior and network topology. The authors proceed to identify many real-world examples of such attacks at both the IP and TCP layers. Several examples of denial of service attacks are also presented, most notably that of resource exhaustion.

In [124], Vigna *et al*. present a systematic, automated framework for testing the resilience of misuse-based intrusion detection systems to evasion attacks. The framework operates by automatically generating and executing mutant exploits, or variations on known attacks, against a testbed composed of vulnerable systems monitored by a set of intrusion detection systems under test. A number of mutation operators, each of which act at either the network, application, or exploit payload levels, are contained within a mutation engine. These mutation operators are responsible for applying semantics-preserving modifications to an exploit, and are deterministic with respect to a given seed value that is supplied by the mutation engine. Mutant exploits are generated from templates, which are exploits that have been annotated with information on how they may be modified by various types of mutation operators. During testing, the mutation engine randomly searches the space of possible mutations, generating and executing mutant exploits against the target systems. An oracle provides feedback to the mutation engine indicating whether a mutant exploit has successfully compromised the target, and whether the IDS has successfully been evaded. The framework was shown to be effective at automatically generating mutant exploits that evaded both Snort and ISS RealSecure while retaining the ability to compromise the target system.

2.6 Server-side web application attack prevention

As with intrusion detection, an extensive literature exists on the detection of web application vulnerabilities. One of the first tools to analyze server-side code for vulnerabilities was WebSSARI [46], which performs a taint propagation analysis of PHP in order to identify potential vulnerabilities, for which runtime guards are inserted. Nguyen-Tuong *et al*. proposed a precise taint-based approach to automatically hardening PHP scripts against security vulnerabilities in [91]. Livshits and Lam [78] applied a points-to static analysis to Java-based web applications to identify a number of security vulnerabilities in both open-source programs and the Java library itself. Jovanovic *et al*. presented Pixy, a tool that performs flow-sensitive, interprocedural, and context-sensitive data flow analysis to detect security vulnerabilities in PHP-based web applications [54]; Pixy was later enhanced with precise alias analysis to improve the accuracy of the technique [55]. A precise, sound, and fully automated technique for detecting modifications to the structure of SQL queries was described by Wassermann and Su in [134]. Balzarotti *et al*. observed that more complex vulnerabilities in web applications can manifest themselves as interactions between distinct modules comprising the application, and proposed MiMoSA to perform multi-module vulnerability analysis of PHP applications [7]. In [14], Chong *et al*. presented SIF, a framework for developing Java servlets that enforce legal information flows specified by a policy language. A syntactic technique of string masking is proposed by Johns *et al*. in [51] in order to prevent code injection attacks in web applications. Lam *et al*. described another information flow enforcement system using PQL, and additionally propose the use of a model checker to generate test cases for identified vulnerabilities [70]. In [6], Balzarotti *et al*. applied

a combination of static and dynamic analysis to check the correctness of web application sanitization functions. Wassermann and Su applied a combination of taint-based information flow and string analysis to enforce effective sanitization policies against cross-site scripting in [135]. Nadji *et al*. propose a similar notion of document structure integrity in [89], using a combination of web application code randomization and runtime tracking of untrusted data on both the server and the browser. Finally, Google's ctemplate [39], a templating language for C++, and Django [23], a Python-based web application framework, include an Auto-Escape feature that allows for context-specific sanitization of web documents, while Microsoft's LINQ [85] is an approach for performing language-integrated data set queries in the .NET framework.

2.7 Client-side web application attack prevention

In addition to server-side vulnerability analyses, much work has focused on client-side protection against malicious code injection. The first system to implement client-side protection was due to Kirda *et al*. In [59], the authors presented Noxes, a client-side proxy that uses manual and automatically-generated rules to prevent cross-side scripting attacks. Vogt *et al*. proposed a combination of dynamic data tainting and static analysis to prevent cross-site scripting attacks from successfully executing within a web browser [127]. BrowserShield, due to Reis *et al*., is a system to download signatures for known cross-site scripting exploits; JavaScript wrappers that implement signature detection for these attacks are then installed into the browser [101]. Livshits and Erlingsson described an approach to cross-site scripting and RSS attacks by modifying

JavaScript frameworks such as Dojo, Prototype, and AJAX.NET in [77]. BEEP, presented by Jim *et al*. in [50], implements a coarse-grained approach to client-side policy enforcement by specifying both black- and white-lists of scripts. Erlingsson *et al*. proposed Mutation-Event Transforms, a technique for enforcing finer-grained client-side security policies by intercepting JavaScript calls that would result in potentially malicious modifications to the DOM [27].

2.8 Functional language security

Several works have studied how the safety of functional languages can be improved. Xu proposed the use of pre/post-annotations to implement extended static checking for Haskell in [138]; this work has been extended in the form of contracts in [139]. Li and Zdancewic demonstrated how general information flow policies could be integrated as an embedded security sublanguage in Haskell in [74]. A technique for performing data flow analysis of lazy higher-order functional programs using regular sets of trees to approximate program state is proposed by Jones and Andersen in [52]. Madhavapeddy *et al*. presented a domain-specific language for securely specifying various Internet packet protocols in [81]. In [34], Finifter *et al*. describe Joe-E, a capability-based subset of Java that allows programmers to write pure Java functions that, due to their referential transparency, admit strong analyses of desirable security properties.

Chapter 3

The Design and Implementation of WEBANOMALY

WEBANOMALY is an advanced anomaly detection system that is designed to protect web applications against attack from malicious clients. WEBANOMALY uses statistical machine learning techniques to automatically build models that characterize various features of web application behavior. These models are then used to detect both known and unknown attacks against web applications. This chapter presents an abstract model for web applications and discusses the design and implementation of WEBANOMALY.

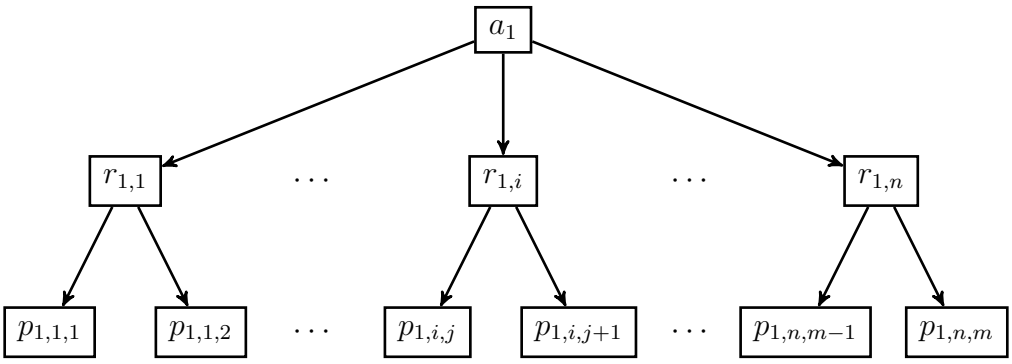


Figure 3.1: Abstract model of the structure of a web application.

3.1 The web application model

As depicted in Figure 3.1, a set of web applications A can generally be decomposed into a tree of *resource paths*, or *components*, R, and named *parameters* P. A web application receives sequences of requests $Q = \{q_1, q_2, \ldots\}$ issued by web clients, and generates in return a sequence of responses $S = \{s_1, s_2, \ldots\}$.

Each request $q \in Q$ can be represented by the tuple $q = \langle a_i, r_{i,j}, P_q \rangle$, where a_i is a web application, $r_{i,j}$ is a resource path associated with the web application, and $P_q \subseteq P_{i,j}$ is a subset of possible parameter name-value pairs that $r_{i,j}$ accepts. The target component of a query, $r_{i,j}$, processes the request and generates a response $s_i = \langle K_q, d_q \rangle$, where K_q is a set of cookies to be instantiated or cleared on the client, and d_q is a document (e.g., HTML, JSON) to be interpreted by the client.

For example, a web application $a_1 =$ `blog.example.com` might be composed of the resources shown in Figure 3.2. Additionally, resource path $r_{i,7}$ might take a set of parameters as part of an HTTP request such as $P_{i,7} = \{p_{i,7,1} = \texttt{oldpw}, p_{i,7,2} = \texttt{newpw}\}$.

$$R_i = \left\{ \begin{array}{l} r_{i,1} = /\texttt{index}, \\ r_{i,2} = /\texttt{article}, \\ r_{i,3} = /\texttt{comments}, \\ r_{i,4} = /\texttt{comments/edit}, \\ r_{i,5} = /\texttt{account/login}, \\ r_{i,6} = /\texttt{account/index}, \\ r_{i,7} = /\texttt{account/password} \end{array} \right\}$$

Figure 3.2: Resources comprising an example web application `blog.example.com`.

A client might issue a query to $r_{i,7}$ of the form

$$q = \left\{ \begin{array}{r} \texttt{blog.example.com}, \\ /\texttt{account/password}, \\ \{(\texttt{oldpwd}, \texttt{foo}), (\texttt{newpw}, \texttt{bar})\} \end{array} \right\}.$$

The web application might issue a response of the form $s = \{\emptyset, \text{“}\texttt{<html>}\ldots\text{”}\}$, indicating that no cookies are to be set, and an HTML document is to be rendered.

For each monitored web application, WEBANOMALY builds an internal set of models. These models mirror the particular abstract structure of each application by analyzing the sequences of queries and responses observed by the system. The following section describes the various components of WEBANOMALY and how these models are constructed.

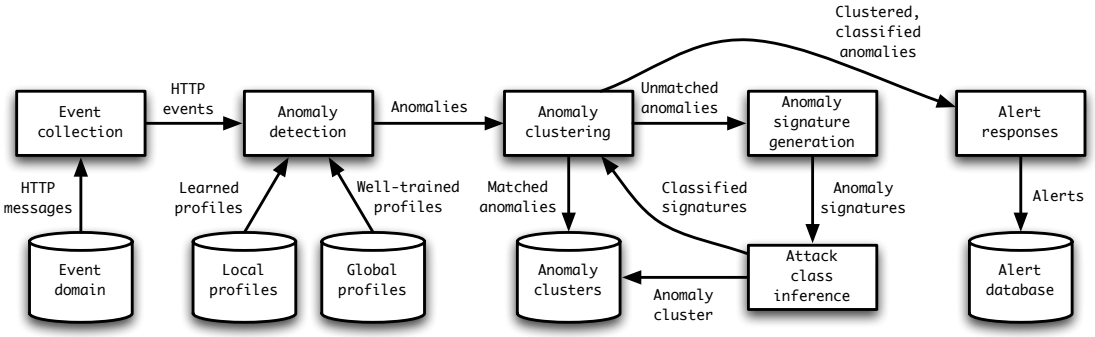


Figure 3.3: Architectural overview of WEBANOMALY.

3.2 The architecture of WEBANOMALY

At a high level, WEBANOMALY can be decomposed into several components. First, an event collection component monitors a network to capture and normalize HTTP messages for analysis. These events are forwarded to an anomaly detection engine that is responsible for both learning the structure and behavior of monitored web applications as well as detecting attacks against the applications. Any detected anomalies subjected to a series of post-processing steps that result in a reduced false positive rate and improved explanatory capability regarding the nature of the anomaly. Alerts can then trigger an action to be executed in response to the anomaly. A graphical overview of the components of WEBANOMALY is presented in Figure 3.3. The following sections describe each of these components in turn.

3.2.1 Event collection

WEBANOMALY performs its analysis on sequences of HTTP requests and responses. Before this can occur, however, raw HTTP messages must be converted into an internal format that is suitable for analysis. Accordingly, WEBANOMALY incorporates an efficient event collection component that can capture HTTP messages either actively or passively.

In passive mode, the event collector captures link-layer packets from the network as they are transmitted between HTTP clients and servers. The collector performs packet defragmentation and stream reassembly for each observed network flow. The resulting stream byte sequences are then made available for HTTP message parsing. Passive mode has the advantage that failures in WEBANOMALY do not affect the normal operation of the set of monitored web applications. Also, misclassifications by the anomaly detector do not have as high of a negative impact on the functioning of monitored web applications, since messages cannot be blocked or modified. On the other hand, passive mode does not allow for the prevention of attacks, since WEBANOMALY only observes network traffic instead of interposing itself between the endpoints.

In active mode, the event collector acts as an HTTP reverse proxy, serving as an intermediary between HTTP clients and servers. Client connections are terminated by the collector, and a connection pool is maintained to a set of back-end web servers. Therefore, both client queries and server responses are directly received as byte sequences by WEBANOMALY. These sequences are then subject to HTTP message parsing for analysis. Active mode has the clear advantage that messages can easily be modified or

blocked in response to detected anomalies. Additionally, this mode of operation allows for the use of network optimization techniques, such as pipelining client requests over established network connections from the server connection pool. Failures or misclassifications, however, have a high impact on the proper functioning of monitored web applications in this mode.

Once a byte sequence representing an HTTP message has been captured, the sequence must be parsed into a high-level representation for analysis. To this end, the event collection component incorporates a robust set of HTTP parsing methods. During parsing, these methods extract a number of features that are examined during the analysis stage. Features extracted from queries include, but are not limited to, the requested path, any parameters contained in both the resource path and message body, HTTP headers such as the content type and length, and the contents of any cookies. Examples of features extracted from responses include relevant headers and parse trees corresponding to the structure and content of several types of documents (e.g., HTML, XML, JSON). Additionally, features from the network layer are extracted from both queries and responses, such as timestamps and endpoint addressing information.

Extensive profiling was performed during the development of the collector (and, indeed, the rest of WEBANOMALY) in order to optimize its performance. For instance, memory copies were eliminated whenever possible to reduce cache pollution and memory bandwidth utilization. Also, many lower-level parsing routines were hand-optimized in assembly to take advantage of advanced processor instructions not utilized by the compiler or to implement efficient string operations specific to HTTP message parsing. As a result, WEBANOMALY is capable of processing significant network loads.

3.2.2 Anomaly detection engine

HTTP messages collected from the network are then forwarded to the anomaly detection engine for analysis. This component is responsible for learning the normal behavior of a set of web applications, and detecting deviations from learned behavior. These specifications of behavior are contained in sets of *models*, where each model uses statistical methods to characterize a specific feature of HTTP messages. These models can be considered to operate in one of two phases: *training* and *detection*.

The training phase consists of an online learning process applied to a sequence of queries and responses. As new resources and parameters are observed, corresponding sets of models are created by the anomaly detection engine. With each subsequent HTTP message, the appropriate models are updated to reflect the behavior of specific features of the message. Once a sufficient number of training samples has been observed, a model or set of models can then switch to the detection phase.

In [66], a fixed-length training phase was proposed. Further experimentation, however, has shown that an appropriate training phase length is highly dependent upon the complexity of modeling a given set of features. Therefore, we have developed an automated method that leverages the notion of *model stability* to determine when a model has observed a sufficient number of training samples to accurately model a feature.

As new training samples are observed early in the training phase, the state of a model typically exhibits frequent and significant change as its approximation of the normal behavior of a feature is updated. Informally, in an information-theoretic sense, the average information gain of each new training sample is high. However, as a model's

state converges to a more precise approximation of normal behavior, its state gradually exhibits infrequent and incremental changes. In other words, the information gain of new training samples approaches zero, and the model stabilizes.

Each model therefore monitors its stability during the training phase by maintaining a history of snapshots of its internal state. Periodically, a model checks if the sequence of deltas between each successive historical state is monotonically decreasing and whether the degree of change drops below a certain threshold. If both conditions are satisfied, then the model considers itself stable and ready to switch to the detection phase.

During the detection phase, sets of models are compared to features of HTTP messages. Using statistical methods, a similarity score is computed between the learned models and observed features – that is, the similarity score is the probability that the observed feature fits the models learned during the training phase. If the similarity between a set of models and the observed feature is less than a given threshold, then the corresponding HTTP message is considered anomalous.

A detailed description of the individual models and learning algorithms is deferred until Section 3.3.

Models that have been trained, or are in the process of being trained, are stored in a local profile database. As Figure 3.3 indicates, a separate *global profile database* containing a disjoint set of models is also available to the anomaly detection component. The function of this database will be discussed in detail in Chapter 5.

3.2.3 Anomaly clustering and characterization

Reports of anomalies from the anomaly engine are forwarded to a set of inter-related components for post-processing. The goal of these components is twofold. First, these components cluster related anomalies in order to reduce the effective false positive rate of the detector. Secondly, a process of attack inference is performed to classify the *type* of attack that an anomaly represents. This gives WEBANOMALY the capability to better explain why an anomaly is malicious.

These components are presented in detail in Chapter 4.

3.2.4 Alert responses

Sets of anomalies, possibly annotated with additional information, are passed from the previous clustering and characterization components to a response component. This component is responsible for performing a set of configurable actions in response to anomalies. In particular, one approach is to log anomalies and prepare reports for analysis by a site security administrator. This approach corresponds to the traditional model of intrusion detection. On the other hand, active responses, such as directly blocking HTTP messages if the detector is in active mode, or inserting firewall rules at the site perimeter, are possible. The active approach corresponds to the so-called intrusion prevention (IPS) approach. Both approaches are implemented for completeness; a discussion, however, of the specific type of response to an anomaly is largely orthogonal to the main focus of this dissertation.

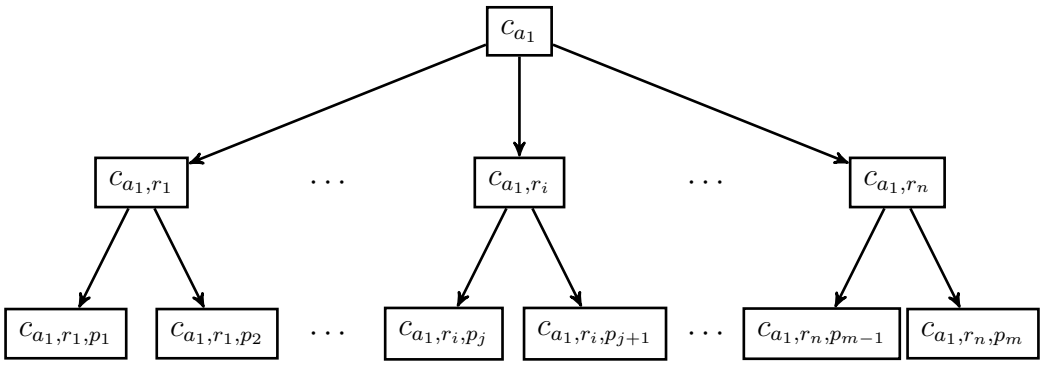


Figure 3.4: The hierarchy of models constructed by WEBANOMALY. Session profiles are created across the entire web application a_1 at the root of the hierarchy. Document profiles are created for each unique resource r_i. Parameter profiles are created for each unique resource and parameter (r_i, p_j).

3.3 Modeling web applications

WEBANOMALY applies sets of models to each of the various features of monitored web applications to be analyzed. Though each of these models utilizes different statistical methods to characterize their normal behavior, some details are common to each model. For instance, each model records the number of training samples observed. Furthermore, each model incorporates a confidence value that denotes the degree to which the model believes it has accurately characterized the feature it is applied to. This confidence value takes values on the interval $[0, 1]$, and is used as a scaling factor on anomaly scores during the detection phase.

A number of models are used to characterize both features of individual HTTP requests as well as sequences of requests from a given client. For individual requests, a set of models are applied to each unique parameter $p_{i,j,k}$ observed during the training phase.

These sets are generically known as *profiles*, and are represented by the tuple

$$c_{p_{i,j,k}} = \left\langle m^{(\text{tok})}, m^{(\text{int})}, m^{(\text{len})}, m^{(\text{char})}, m^{(\text{struct})} \right\rangle .$$

For sequences of HTTP requests $Q = \{q_1, q_2, \ldots\}$ from a given client, a profile consisting of a pair of models is applied:

$$c_{a_i} = \left\langle m^{(\text{int})}, m^{(\text{sess})} \right\rangle .$$

Finally, a profile is applied to characterize the structure of responses $r_{i,j}$ generated by the monitored web applications, where

$$c_{r_{i,j}} = \left\langle m^{(\text{doc})} \right\rangle$$

Figure 3.4 displays the hierarchy of profiles created for each web application. Session profiles are created across an entire web application a_1 at the root of the hierarchy. Document profiles are created for each unique resource r_i. Finally, parameter profiles are created for each unique resource and parameter (r_i, p_j).

In the following, the details of each of the individual models is described, as well as how each set of models is composed during the detection phase in order to derive a final anomaly score.

```
<select name="role">
<option value="user"/>
<option value="devel"/>
<option value="admin"/>
</select>
```

Figure 3.5: Example of an HTML `<select/>` input field that can be characterized by a token model.

3.3.1 Token model

The token model $m^{(\text{tok})}$ characterizes features that take values drawn from a discrete set of values. For instance, consider an HTTP request parameter that corresponds to an HTML `<select/>` input field, shown in Figure 3.5. It is clear that the `role` parameter to the web application should only take one of three values from the set $\{\texttt{user}, \texttt{devel}, \texttt{admin}\}$. Indeed, this can be viewed as an informal specification of the legitimate behavior of the parameter by the developers of the web application.

It is often the case, however, that hidden parameter values can enable additional functionality in a web application that should not be accessed by unauthorized clients. For example, the `role` parameter might also accept the value `test`, activating a debugging interface that exposes sensitive functionality. Token models are particularly well-suited to detecting such attacks, where the set of legitimate values is relatively small and is explicitly encoded by the developers.

Training phase. During the training phase, the set of unique observed values V for a given feature is recorded. Then, at periodic intervals a statistical test is performed to determine the correlation between the number of unique values and the total number of

samples observed since the beginning of the training phase. In [66], an *ad hoc* test was used to calculate the correlation coefficient between these two values. WEBANOMALY has since eschewed this test in favor of the Kendall τ rank correlation coefficient [57], a non-parametric test to measure the degree of correspondence between two rankings. The test is defined as

$$\tau = \frac{n_c - n_d}{\frac{1}{2}n\left(n-1\right)},$$

where n_c is the number of *concordant* pairs, n_d is the number of *discordant* pairs, and n is the total number of pairs. In this application, a concordant pair is assumed if a new value was observed at a training step, and a discordant pair is assumed otherwise. Therefore, if unique values are often observed at each training step, the resulting correlation will be high, and the confidence of the token model will be set to zero. Otherwise, if the correlation is low, the token model will be used to characterize the feature, with a confidence value set to the normalized correlation coefficient.

Detection phase. During the detection phase, the probability of an observed value v is simply

$$\Pr\left(v\right) = \begin{cases} 0 & \text{if } v \notin V \\ 1 & \text{otherwise} \end{cases}$$

3.3.2 Integer model

Many features of web applications can be characterized as a set of integers drawn from an unknown distribution. For instance, HTTP request parameters may expect strings encoding integer literals as values. The lengths of parameter values are another feature that manifests as an integer distribution. Attacks against web applications can often manifest themselves as deviations from these distributions, however. For example, a buffer overflow or code injection attack may result in a parameter length that is significantly longer than normal values. WEBANOMALY incorporates an integer model $m^{(\text{int})}$ that characterizes the normal behavior of integer distributions, allowing for the detection of such attacks.[1]

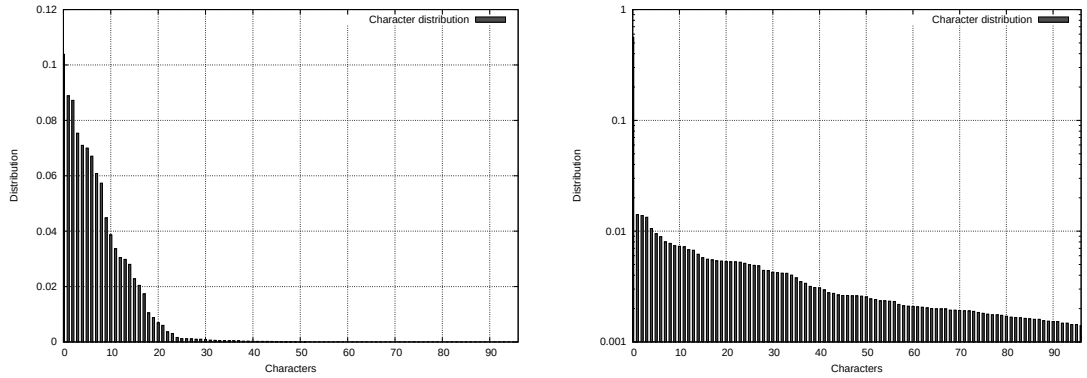
Training phase. During the training phase, the sample mean μ and variance σ^2 are incrementally computed from observed values. At periodic intervals, the stability of these statistics are checked as described in Section 3.2.2.

Detection phase. The probability of an observed integer sample v is calculated during the detection phase through an application of the Chebyshev inequality, where

$$\Pr\left(|x-\mu|-|v-\mu|\right) < \Pr\left(v\right) = \frac{\sigma^2}{\left(v-\mu\right)^2}.$$

The Chebyshev inequality is a non-parametric test that uses the sample mean and variance to establish a bound upon the probability of the deviation of an observed value

[1] Note that the length model $m^{(\text{len})}$ is simply an application of $m^{(\text{int})}$ to the length of strings.



(a) Character distribution for normal English text. (b) Character distribution for a buffer overflow.

Figure 3.6: Character distribution models for both legitimate and malicious values.

from the sample mean. The bound that is computed is weak, thereby providing a measure of generalization from the training data. That is, only significant deviations from the training set are considered to be anomalous.

3.3.3 Character distribution model

Assuming an ASCII or UTF-8 character encoding, HTTP request parameters can be considered as sequences of (largely) 8-bit character values. Parameter values typically do not exhibit a uniform distribution of characters, however. Rather, it is often the case that printable characters such as letters, numbers, and punctuation solely comprise the legitimate set of parameter values. Indeed, well-known distributions of character frequencies exist for English and many other languages.

On the other hand, attacks often include non-printable characters, in the case of buffer overflow attacks, or special characters that do not often appear in legitimate values, as in the case of XSS or SQL injection attacks. Furthermore, characters can be distributed

in a significantly different manner than that for legitimate values. Figure 3.6 shows two example character distributions. In Figure 3.6a, a standard frequency histogram for English text can be observed. In Figure 3.6b, however, the histogram for a buffer overflow clearly indicates a long-tailed distribution that differs significantly from English text.

Therefore, WEBANOMALY includes a character distribution model $m^{(\text{char})}$ that characterizes the normal distribution of characters comprising various web application features.

Training phase. During the training phase, the frequencies of individual characters are recorded. At periodic intervals, the *idealized character distribution* (ICD) of the modeled feature is generated. The ICD is constucted by taking the mean of the frequency of each character observed, normalizing the mean to the interval $[0, 1]$, and ranking these frequencies from highest to lowest. Because all individual character distributions sum to unity, their average will do so as well, and the ICD is therefore well-defined. A sliding window of ICDs is maintained during the course of the training phase, and when the ICDs converge to a fixed point, the character distribution model switches to the detection phase.

Detection phase. In this phase, the ICD of each observed feature value v is constructed. The probability of v is then given by the similarity between the ICD learned during the training phase and the ICD generated from v. This similarity is calculated using the Pearson χ^2 goodness-of-fit test. The χ^2 test requires that the function domain be discretized into a small number of intervals, and it is preferable that all intervals

contain "some" elements.[2] Therefore, given m intervals and binned ICDs O and E for the observed and learned ICDs respectively, the probability of v is given by

$$\Pr(v) \equiv \chi^2 = \sum_{i=0}^{i<m} \frac{(O_i - E_i)^2}{E_i},$$

where the computed χ^2 value is used as an index into a predefined probability table.

3.3.4 Structure model

Although many attacks are sufficiently different from normal behavior to be reliably detected using the previously described models, some attacks manifest themselves as feature values that are "close" to normal behavior. For instance, some attacks might not be significantly longer than legitimate parameter values, or an attack might be encoded to include only characters corresponding to English text. In these cases, a more powerful model is needed to characterize the feature in order to reliably discriminate between normal and malicious values. WEBANOMALY includes a structure model $m^{(\text{struct})}$ for this purpose.

The structure model is an instance of a Hidden Markov Model (HMM), a probabilistic finite state automaton that encodes a grammar that can generate a superset of the legitimate feature values observed during the training phase. An HMM is represented by the tuple

$$m^{(\text{struct})} = \langle \mathbb{S}, \mathbb{E}, \delta, P \rangle ,$$

[2]The literature suggests that at least five elements is sufficient in most cases.

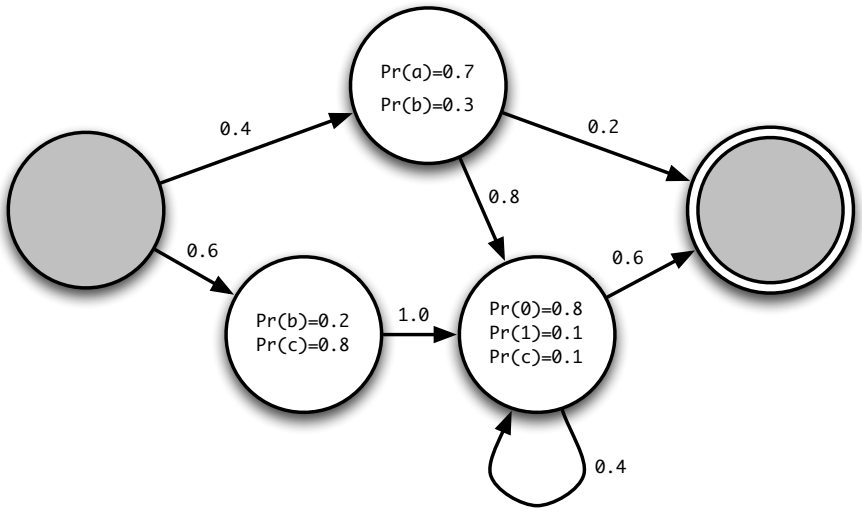


Figure 3.7: Example of a structure model HMM. Here, the start state has no emissions and indegree 0. The final state has no emissions and outdegree 0; it is indicated with a double border. The remaining nodes indicate a probability distribution over the symbols that may be emitted at each node, and arcs are labeled with their transition probabilities. The set of arcs directed from a node form a transition probability distribution from that node.

where $\mathbb{S}$ is the set of states, $\mathbb{E}$ is the set of symbol emissions, $\delta : \mathbb{S} \times \mathbb{E} \mapsto \mathbb{S}$ is a transition function, and $P : \mathbb{S} \times \mathbb{S} \times \mathbb{E} \mapsto [0, 1]$ is a probability distribution over the sets of states, transitions, and emissions. A graphical representation of a structure model is shown in Figure 3.7.

Training phase. The process used to construct a structure model is an online algorithm due to Stolcke [116]. Initially, the HMM is composed of unconnected start and end states s_0 and s_1. For each observed feature value, the HMM is updated to reflect the observed sequence of symbols comprising the value. That is, starting from s_0, a check is performed to determine whether a transition exists from the current state to a target state that can emit the next symbol. If such a transition exists, the traversal counts for

both the existing transition and symbol at the target state are incremented. Otherwise, a new target state, emission, and transition from the current state to the target state are created.

At regular intervals during the training phase, a process of state merging is performed on the current HMM. The goal of merging states is to induce generalization in such a way that the *a posteriori* probability of the HMM given the training data is maximized. This problem can be formulated using the celebrated Bayesian theorem

$$\Pr(M \mid D) = \frac{\Pr(D \mid M) \Pr(M)}{\Pr(D)},$$

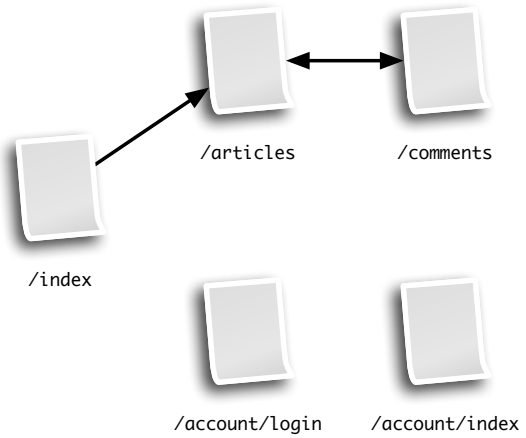
where M is the HMM, D is the training data, $\Pr(D \mid M)$ is the likelihood of the data given the model, $\Pr(M)$ is the prior probability of the model, $\Pr(D)$ is the probability of the data, and $\Pr(M \mid D)$ is the posterior probability of the model given the data. At each step during the merging process, the pair of states whose merge would result in an HMM with a higher posterior probability are selected. If no such pair exists, the merging process stops. The training phase continues to update the HMM and periodically merge states until the posterior probability of the HMM converges to a fixed point.

One important point to note is that the sequence of symbols derived from a parameter value typically does not have a one-to-one correspondence with the actual characters comprising that value. This is primarily due to the fact that the computational complexity of evaluating an HMM over a symbol alphabet, where $|\mathbb{E}|| \gg 8$ would be prohibitive in practice. Therefore, prior to applying Stolcke's algorithm, a process of symbol compression is performed, where characters are mapped to a small number of symbols.

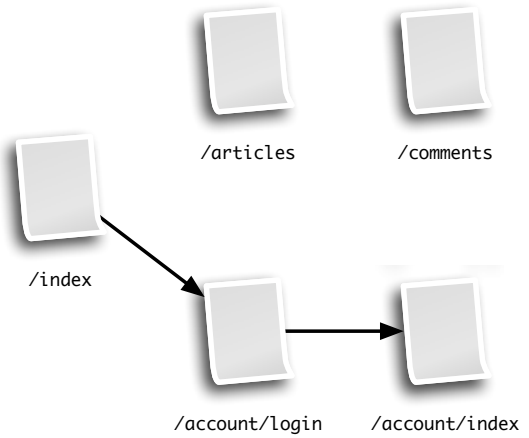
In [66], a fixed mapping was defined, where contiguous sequences of lowercase letters were mapped to the symbol a, uppercase letters were mapped to the symbol A, and so forth. This mapping, however, was found to induce a degree of generalization that was undesirable in some cases (i.e., the resulting HMM would generalize to the point where it was unable to accurately classify samples).

Consequently, a new algorithm was devised to construct a dynamic mapping between characters and symbols. This algorithm begins by first calculating the digram probabilities of contiguous pairs of characters observed during the training phase. Once a sufficient number of pairs have been observed – that is, the digram probability distribution has converged – a greedy criterion is used to distribute characters among a fixed-size symbol set. Pairs of characters are processed in order of decreasing probability, and assigned to different symbols. This heuristic results in the retention of a greater amount of structural information, while at the same time reducing the computational complexity of the generated HMM.

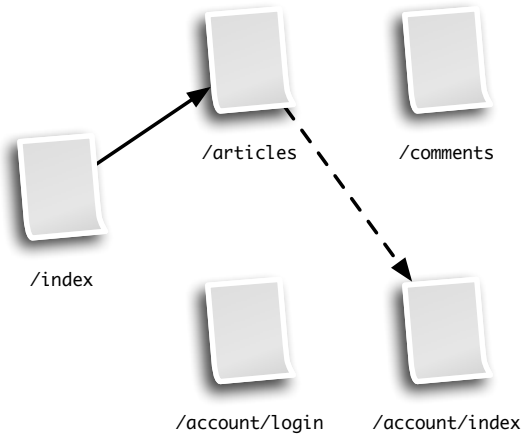
Detection phase. During the detection phase, the probability of an observed value v is calculated by applying the Viterbi path approximation algorithm [126] to the corresponding symbol sequence. This well-known algorithm calculates the most likely sequence of hidden states that can generate v, from which the probability of v can be derived.



(a) Example session structure for a user.



(b) Example session structure for an administrator.



(c) Example session structure for a privilege escalation attack.

Figure 3.8: Example session structures. In (c), the dotted arc represents an anomalous transition from the unprivileged resource `/articles` to the privileged resource `/account/index`.

3.3.5 Session model

As opposed to the previous models, which are exclusively focused on individual queries, the session model $m^{(\text{sess})}$ is applied to sequences of queries issued by HTTP clients. The intuition behind this model is that paths through a web application typically possess a regular structure. For instance, consider the example web application resources shown in Figure 3.2. A user of the website might issue requests with the structure shown in Figure 3.8a, where the index page is initially loaded, and a sequence of articles and comments are viewed. The owner of the web application might have a session structure shown in Figure 3.8b, where an administrative interface is accessed after having successfully authenticated. On the other hand, the session structure shown in Figure 3.8c might correspond to a successful privilege escalation attack, where a user gains direct access to the administrative interface, bypassing the authentication resource.

Accordingly, a $m^{(\text{sess})}$ is an adaptation of $m^{(\text{struct})}$ to the domain of web application resources, where resources are mapped to symbols. Additionally, several parameters of the state merging criterion are modified in order to reduce the amount of generalization induced by the algorithm. Though this necessarily increases the complexity of the resulting HMM, this is compensated for by the fact that a single instance of $m^{(\text{sess})}$ is typically applied for each web application.

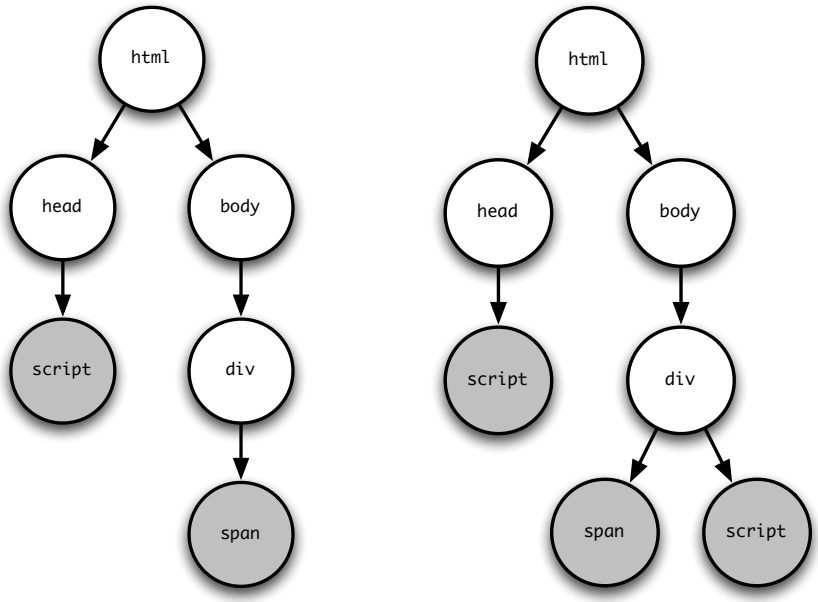
3.3.6 Document model

The models discussed to this point have been concerned solely with characterizing features of HTTP requests. WEBANOMALY also includes a model to capture the normal

behavior of documents generated by monitored web applications. Many documents created by web applications conform to a regular structure. For instance, HTML documents often have common header and footer sections that do not change significantly from page to page. Also, JSON documents that are returned to clients in response to an AJAX request often follow a particular structure.

In the case that a successful attack has occurred, however, the structure of documents generated by the web application can change drastically. If a data breach has occurred as part of a SQL injection, for example, an HTML document might be returned that contains a large table containing sensitive data such as credit card numbers or Social Security numbers where none were present before. Or, in the case of a XSS reflection to a client, client-side code might appear in the body of the document where none had been previously observed. To detect such malicious behavior, WEBANOMALY incorporates a document structure model $m^{(\text{doc})}$ that attempts to characterize both the normal structure of documents as well as the locations of sensitive information and client-side code within documents.

Training phase. During the training phase, WEBANOMALY parses various types of documents observed in response to requests issued by clients of monitored web applications. Supported document types currently include HTML, XHTML, XML, and JSON documents. The resulting parse trees are then pruned according to the following algorithm. For each tree, a depth-first search is performed for nodes that contain identifiably sensitive data, such as Social Security numbers or credit cards validated using the standard Luhn check. In addition, any nodes that contain client-side code such as



(a) Legitimate document. (b) Document containing reflected XSS attack.

Figure 3.9: Document structure models, with security-relevant nodes highlighted. In (a), the legitimate document contains client-side code in the `<script/>` tag, and sensitive data in the `<span/>` tag. In (b), a malicious `<script/>` tag has been inserted into the body of the document.

HTML `<script/>` tags or DOM event handlers are identified. These security-relevant nodes, and all parent nodes to the root of the document, are retained; the remaining nodes are discarded. Additionally, the number of nodes containing sensitive data is recorded. For each successive parsed response corresponding to a given resource, the partial tree of security-relevant nodes for a response are merged into the existing tree. Also, the mean and variance of the number of nodes containing sensitive data is incrementally computed. The merged tree, including node annotations corresponding to security-relevant features, and the sensitive data statistics comprise $m^{(\text{doc})}$. As for the previously-described models, when this merged tree converges to a fixed point, the model switches to the detection phase.

Example structures for HTML documents are shown in Figure 3.9. In Figure 3.9a, a partial tree has been constructed to indicate the security-relevant nodes `<script/>` and `<span/>`. An example of a reflected XSS attack is shown in Figure 3.9b, where a malicious `<script/>` node has been inserted into the body of the document.

Detection phase. Given an observed document v, the probability of its partial tree obtained as described above is given by

$$\Pr(v) = w_c \left(1 - \frac{m_c}{n_c}\right) + w_d \frac{\sigma_d^2}{(n_d - \mu_d)^2},$$

where m_c is the number of client-side code instances not found in $m^{(\text{doc})}$; n_c is the total number of instances of client-side code; n_d is the total number of nodes containing sensitive data; μ_d and σ_d^2 are the mean and variance, respectively, of nodes containing sensitive data; and w_c, w_d are weights such that $w_c, w_d \geq 0$ and $w_c + w_d = 1$.

3.3.7 Model composition

Each preceding model outputs a probability score on the interval $[0, 1]$ that indicates the likelihood of a sample given the model. Since, however, sets of models are generally applied to features of web applications, it is necessary to combine the scores from a profile's constituent models in order to derive a final probability score. In many ensemble learning methods, which WEBANOMALY could be considered to employ, a simple majority voting scheme or weighted summation of the model outputs is performed.[3] These

[3]For instance, a naïve Bayesian network is an example of such a model combination.

approaches, while lightweight, do not take into account complex interdependencies that exist between each of the models comprising a profile.

Therefore, WEBANOMALY instead applies non-naïve Bayesian networks to compose the outputs of multiple models [65]. Bayesian networks are directed acyclic graphs (DAG) that allow for probabilistic reasoning under uncertainty. The DAG is composed of nodes representing discrete random variables. Each node contains the states of the random variable it represents and a conditional probability table (CPT). A CPT specifies the joint probability distribution of a node's random variable given the random variables of the node's parents. Links between nodes indicate causal dependence; that is, the variable represented by a child node is causally dependent upon the variables represented by the node's parents. Nodes that are unconnected are causally independent. Additionally, nodes comprising a Bayesian network are either *evidence* nodes, which indicate some measurable value, or are *hypothesis* nodes, which correspond to values that cannot directly be observed. Bayesian networks allow one to obtain the probability of the hypothesis variable given the evidence. In our application, the evidence nodes are the outputs and various states of the models comprising a profile, and the hypothesis node indicates whether an attack has been detected given the state of the models.

Note that naïve Bayesian networks have a tree structure with the hypothesis node at the root and a set of evidence nodes at a depth of 1. This structure corresponds to a weighted summation of the individual evidence nodes. The Bayesian networks used by WEBANOMALY contain causal links between evidence nodes to indicate model interdependence, and are therefore not equivalent to a weighted summation.

Figure 3.10 presents an example Bayesian network used by WEBANOMALY to classify

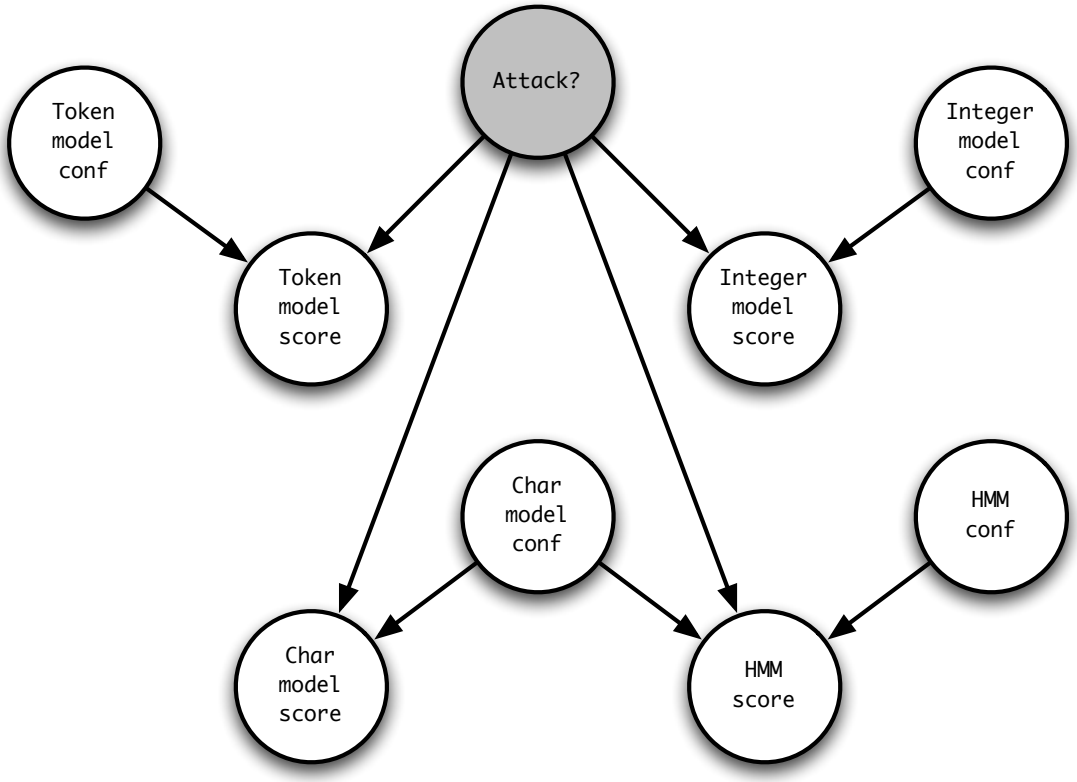


Figure 3.10: Example Bayesian network model composition for a parameter profile. In this example, model probability scores and confidence values are represented by evidence nodes. The highlighted hypothesis node represents the probability of an attack given the evidence. Note that the structure model probability score is dependent upon both the confidence of the HMM as well as of the character distribution.

parameter values. Here, evidence nodes are instantiated for each model probability score and confidence value comprising a parameter profile. The highlighted hypothesis node represents the probability of an attack given the evidence. Note that the structure model probability score is dependent upon both the confidence of the HMM as well as of the character distribution. With the use of Bayesian networks, inter-model dependencies such as the one shown are easily modeled in a systematic way.

3.4 Conclusions

WEBANOMALY employs a black-box approach to protect vulnerable web applications. Sets of statistical models, known as *profiles*, are applied to various features of web applications throughout the hierarchy of the abstract model shown in Figure 3.1. Additionally, Bayesian networks are used to probabilistically reason about the low-level model outputs in a strong, theoretically sound way. Using these techniques, WEBANOMALY is able to detect attacks with high accuracy and low performance overhead.

Achieving these results, however, required the author to overcome several fundamental challenges. The following chapters of this dissertation introduce each of these challenges in turn and how they were addressed.

Chapter 4

Clustering and Characterization of Anomalies

As Axelsson demonstrated in a seminal paper [4], the false positive rate produced by an anomaly detection system is the dominating factor as to the usability of that system in practice. Put succinctly, if the false positive rate of an anomaly detector is too high, then, as a consequence of the low prior probability of attacks, the posterior probability that an alert represents a true attack is low. Therefore, security researchers have focused much effort on reducing the false positive rate of intrusion detection systems. Due to the complexity of modeling features of modern software behavior, however, some amount of false positives is inevitable. Indeed, the relatively high rate of false positives produced by anomaly-based IDSs remains a major obstacle to their widespread adoption.

A secondary problem relating to anomaly detection systems is that, even assuming that

an alert does in fact represent a true positive, an alert generated by an anomaly detection system does not provide a significant amount of actionable information. Misuse-based systems can typically provide extremely fine detail as to the type of vulnerability and attack that an alert corresponds to. In contrast, because anomaly-based systems do not explicitly model malicious behavior, their analysis of the nature of an anomaly is limited to generic statements such as "parameter value too long" or "parameter value contains bad character(s)."

As a result, a major area of research regarding WEBANOMALY has been on addressing these two problems. Several features of the system that have already been discussed, such as the composition of multiple models for a given feature, are intended to address the issue of false positive reduction. This chapter, however, presents a complementary approach to both reducing the false positive rate of an anomaly detector as well as increasing its ability to analyze malicious behavior: *anomaly signatures*.

The key intuition behind the concept of anomaly signatures is that similar anomalies are likely to represent similar attacks in the case of true positives, or to represent similar misclassifications in the case of false positives. For instance, a set of models may share a common defect, such that false positives are generated in the same way in response to specific events. Alternatively, an attacker may probe a website for a specific vulnerability, causing a group of related alerts. By clustering similar anomalies, it is therefore possible to reduce the number of alerts, and thereby false positives, reported to security administrators. Additionally, through a process of *attack class inference*, these groups of anomalies are labeled according to the type of attack they represent, significantly

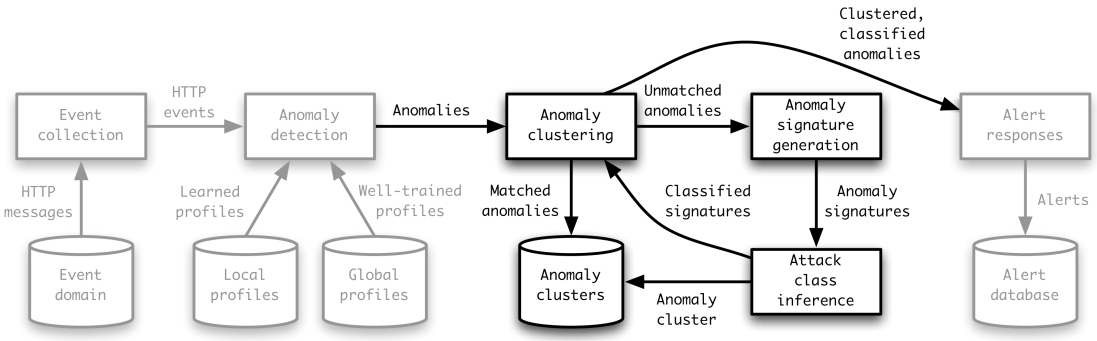


Figure 4.1: Architectural overview of WEBANOMALY, with anomaly characterization and clustering components highlighted.

improving the explanatory power of the resulting anomaly detection system.[1]

4.1 Architecture

Figure 4.1 presents an overview of the high-level architecture of WEBANOMALY. Many of the components comprising the system have already been introduced in Section 3.2. In the following, however, we discuss the high-level interactions between components that are responsible for anomaly clustering and characterization. A detailed discussion of the operation of each component is discussed in the following sections.

4.1.1 Anomaly clustering

Anomalies that have been generated by the anomaly detection component are forwarded to the *anomaly clustering* component. This component maintains a database of

[1] An earlier version of this work was presented in [106].

previously generated anomaly signatures, against which new anomalies are compared. If a match is found, the anomaly is associated with the existing anomaly signature. Otherwise, the anomaly is passed to subsequent components for further processing.

4.1.2 Anomaly signature generation

Anomalies that are not matched against any existing anomaly signature are forwarded to the *anomaly signature generation* component. The task of this component is to construct a generalized anomaly signature that is based on parameters extracted from a particular anomaly. The generated signature is used to match further manifestations of similar anomalies.

Note that a distinction must be made between *misuse signatures* and *anomaly signatures*. In our nomenclature, a misuse signature describes a specific instance of a known attack, while an anomaly signature consists of a generalized description of a deviation from models of normal behavior that is dynamically extracted from an anomaly.

4.1.3 Attack class inference

In many cases, an anomaly signature can be further classified as belonging to one of several broad, generic classes of known attacks against web-based applications. In our system, these classes curently include buffer overflows, cross-site scripting (XSS), SQL injection, command injection, and directory traversal. The task of the attack class inference component is to select a label representing the type of attack that an anomaly

represents. This label is then applied to the corresponding anomaly signature.

This process of attack classification results in semantically rich anomaly alerts that are more informative to security administrators and web application developers. These alerts not only pinpoint the vector through which the application is being attacked (e.g., a specific resource and parameter), but, in addition, identify the nature of the attack as well. This can assist the security administrator or application developer in more quickly mitigating the vulnerability, as the nature of the attack itself generally suggests the required steps for remediation.

4.2 Anomaly clustering and signature generation

The central object of the anomaly clustering and characterization process is the anomaly signature. An anomaly signature is composed of a set of parameterized models corresponding to the profile that generated an anomaly. That is, when a profile generates an anomaly, the parameters for each of its constituent models are extracted to create an anomaly signature. As an example, consider a profile for a parameter to a web application resource

$$c = \left\langle m^{(\text{tok})}, m^{(\text{int})}, m^{(\text{len})}, m^{(\text{char})}, m^{(\text{struct})} \right\rangle .$$

Let $p\left(m^{(\cdot)}\right)$ be a set of parameters associated with a given model. Then, we define an *unlabeled anomaly signature* for parameter profiles to be

$$g = \left\langle p\left(m^{(\text{tok})}\right), p\left(m^{(\text{int})}\right), p\left(m^{(\text{len})}\right), p\left(m^{(\text{char})}\right), p\left(m^{(\text{struct})}\right)\right\rangle.$$

Anomaly signatures for session and document models are defined similarly.[2]

In order to cluster anomalies, a similarity measure must be defined between anomaly signatures. To that end, we define an anomaly signature distance metric between two anomaly signatures g_1, g_2 as

$$d_g = \frac{1}{W}\sum_{i=1}^{P} w_i \cdot d_{m^{(\cdot)}}\left(p\left(m_{1,i}^{(\cdot)}\right), p\left(m_{2,i}^{(\cdot)}\right)\right),$$

where $d_{m^{(\cdot)}}$ is a model-specific similarity measure, $m_{1,i}^{(\cdot)}$ is the i$^{\text{th}}$ model of signature g_1, w_i is a weight associated with model $m_{j,i}^{(\cdot)}$, $W = \sum_{i=1}^{P} w_i$, and P is the number of model parameter sets contained in g_1, g_2.

The following sections describe the individual similarity measures $d_{m^{(\cdot)}}$ for each type of model.

[2]Note that the anomaly signature generation process is not driven by examples of known attacks, but rather by the extraction of parameters from a set of anomaly models. Therefore, these anomaly signatures are not derived from (or associated with) a specific exploitation technique.

4.2.1 Token model

The token model determines the probability of an observed value v given a learned token set V as

$$\Pr(v) = \begin{cases} 0 & \text{if } v \notin V \\ 1 & \text{otherwise} \end{cases}.$$

During the signature generation phase, an anomalous value v is extracted such that $p\left(m^{(\text{tok})}\right) = \{v\}$. Consequently, the distance metric between token model parameters is given by

$$d_{m^{(\text{tok})}} = \text{lex}(v_1, v_2),$$

where lex is a function to calculate lexicographic similarity between sequences of characters normalized to the interval $[0, 1]$. Note that lex can be tuned to achieve different levels of sensitivity to variations in the values of anomalous tokens. For instance, lex may use simple metrics such as the Hamming or Levenshtein distances. Alternatively, a higher-level phonetic distance such as the Soundex algorithm may be used.

In the degenerate case, lex always returns true and the resulting metric would consider all tokens not contained in V to be equivalent. In the current implementation, however, lex is a simple string equality test.

4.2.2 Integer model

The integer model represents the unknown distribution of a set of integers as the sample mean μ and variance σ^2, and the probability of an observed value v is

$$\Pr\left(|x-\mu|-|v-\mu|\right) < \Pr\left(v\right) = \frac{\sigma^2}{\left(v-\mu\right)^2}.$$

These two values, as well as the anomalous value itself, are extracted as model parameters – that is, $p\left(m^{(\text{int})}\right) = \{v, \mu, \sigma^2\}$. The distance metric between integer model parameters is then defined as

$$d_{m^{(\text{int})}} = \left| \frac{\sigma^2}{\left(v_1-\mu\right)^2} - \frac{\sigma^2}{\left(v_2-\mu\right)^2} \right|.$$

4.2.3 Character distribution model

The character distribution model stores the idealized character distribution (CD) of a given feature. During the detection phase, it applies the Pearson χ^2 test to determine the probability of an observed sample v. If the observed distribution is found to be anomalous, the parameters of the character distribution model are extracted in one of two ways, depending on the nature of the anomaly.

If the observed character distribution exhibits a sharp drop-off indicating the dominance of a small number of characters, a configurable number n of the character values that

dominate the distribution are extracted. That is, the set

$$p\left(m^{(\text{char})}\right) = \{(c_1, f_1), (c_2, f_2), \ldots, (c_n, f_n)\}$$

is constructed, where c_i is the i$^{\text{th}}$ dominating character value and f_i is the corresponding relative frequency. Then, the distance metric between character distribution parameters is

$$d_{m^{(\text{char})}} = \left(1 - \frac{|n_1 - n_2|}{|n_1 + n_2|}\right) \sum_{i=1}^{\max(n_1,n_2)} |f_{1,i} - f_{2,i}| .$$

If, however, the observed character distribution is close to uniform, the distance metric becomes a test for uniformity. That is,

$$d_{m^{(\text{char})}} = \sum_{i=1}^{\max(n_1,n_2)} |f_{1,i} - f_{2,i}| .$$

Note that these two different techniques are necessary to accomodate common attacks that manifest themselves as character distribution anomalies with disparate characteristics.

4.2.4 Structure model

Recall that the structure model learns a probabilisitic grammar that can generate a superset of the feature values observed during the training phase. If an observed value v is determined to be anomalous, the symbol sequence corresponding to v is first extracted.

Let s denote such a sequence. Then, a linear scan of s is performed to identify whether any of the symbols were underivable – that is, whether, at any point i in the sequence, a transition was not available to a state emitting s_{i+1}. If one is found, the sequence is truncated to obtain $s' = \{s_1, \ldots, s_{i+1}\}$. Here, the use of a sequence prefix is motivated by the observation that repeated attacks against a particular web application feature often exhibit similar prefixes in the malicious values comprising the attacks. Otherwise, if no such symbol can be found, let $s' = s$. The parameter set for the structural model is then $p\left(m^{(\text{struct})}\right) = \{s'\}$.

Accordingly, we define the distance metric between structure model parameters as

$$d_{m^{(\text{struct})}} = \text{lex}\left(s'_1, s'_2\right),$$

where lex is a lexicographic comparison function as in the case of the token model.

4.2.5 Session model

The session model characterizes legitimate flows through a web application by considering sessions as sequences of resource symbols. Then, legitimate flows are represented by constructing a probabilistic automaton that can generate the flows observed during training, as well as similar flows. This approach is similar to that employed by the structure model. In this case, however, the prefix of a malicious flow is generally less interesting than the transition to a protected resource that often exemplifies the type of attack a session model is designed to detect.

Therefore, if an observed session flow v is observed to be anomalous, the corresponding symbol sequence s is extracted as in the case of the structure model. In contrast, however, if a linear scan of s identifies an underivable symbol s_{i+1}, only the pair of resources representing the malicious transition is extracted; that is, we let $s' = \{s_i, s_{i+1}\}$. If, on the other hand, no such malicious transition is found, then $s' = s$ as in the case of the structure model. The parameter set for the session model is then $p\left(m^{(\text{sess})}\right) = \{s'\}$.

Similarly, we define the distance metric between session model parameters as

$$d_{m^{(\text{sess})}} = \text{lex}\left(s'_1, s'_2\right).$$

4.2.6 Document model

Recall that the document model learns a superset of legitimate *partial parse trees* of a variety of documents, where only nodes that are deemed security-relevant, and the parent nodes to the root of the document, are retained. When an observed document v is determined to be anomalous, a subgraph of its partial parse tree is extracted, where only those security-relevant nodes and their parents are retained. Let v' denote this sub-graph. Then, the parameter set for document models is $p\left(m^{(\text{doc})}\right) = \{v'\}$.

To derive a similarity metric between document model parameter sets, we perform a fuzzy subgraph isomorphism test by determining the number of distinct paths from the root to a security-relevant node match between two trees v'_1, v'_2. Let m be the number of paths not found in both trees, and let n be the total number of distinct paths in both

trees. Then, the distance metric between document model parameters is simply

$$d_{m^{(\text{doc})}} = 1 - \frac{m}{n}.$$

4.3 Attack class inference

While providing the ability to detect previously unknown attacks, anomaly detection is, in general, not able to provide a concrete explanation of what an attack represents with respect to the target application. This is a general limitation of anomaly detection approaches and often confuses security administrators when they have to analyze alerts that state only that some feature does not match established profiles.

We have observed, however, that many well-known classes of attacks share common high-level representations. Therefore, whenever an anomaly is detected, various techniques can be used to attempt to infer the type of an attack and provide additional information to security administrators.

Consequently, WEBANOMALY includes an attack class inference component to determine the *class* of attack that an anomaly generated by the system represents. Here, *ad hoc* heuristics are used to classify anomalous values as belonging to one of several categories of attacks. These categories include buffer overflows, cross-site scripting (XSS), SQL injection, and directory traversal.

As noted in Section 4.1.2, the attack class inference process is fundamentally different from the matching of "traditional" intrusion detection signatures. This is due to the fact

that the inference process is only applied to values that have *already been identified as anomalous*, while misuse signatures are applied to the entire event being analyzed. As a consequence, the attack class inference technique can be less precise without incurring the risk of classifying benign portions of an event as malicious. Additionally, we note that attack inference is independent of the derivation of anomaly signatures, and does not always produce a definitive classification.

In the following, we discuss each of the heuristics used to label an anomaly as belonging to one of a set of attack classes. Once a set of heuristics have been applied, the *labeled anomaly signature* with label l is denoted by

$$g' = \left\langle p\left(m^{(\text{tok})}\right), p\left(m^{(\text{int})}\right), p\left(m^{(\text{len})}\right), p\left(m^{(\text{char})}\right), p\left(m^{(\text{struct})}\right), l\right\rangle.$$

4.3.1 Directory traversal

Directory traversal attacks are essentially attempts to gain unauthorized access to files that are not intended to be exposed by a web application or web server. These attacks are accomplished by escaping the web server document root using "..." to reference parent directories. These attacks are somewhat unique in that a small set of characters is involved in their execution, namely "." and "/". Accordingly, the heuristics for detecting directory traversals are only activated if either the character distribution returns a dominating character set C where $C \bigcap \{., /\} \neq \emptyset$, or if the structural inference model returns a violating character-compressed string with a final underivable character of "." or "/". To infer the presence of a directory traversal attack, the heuristic scans

the anomalous feature for a substring derivable by the regular grammar represented by `(/|\.\.)+`.

For example, suppose that an anomaly is generated from a parameter value of "`cat ../../../../../etc/shadow`". In this case, the character distribution model might identify an anomalous number of "`.`" and "`/`" characters, and, in addition, the structural model might detect a violation of the attribute structure. As a consequence, the directory traversal attack class inference heuristic is applied to the anomalous attribute value. The heuristic determines that the value matches the regular expression `(/|\.\.)+`, and the attack is labeled as a directory traversal attack.

4.3.2 Cross-site scripting

XSS attacks involve the injection of client-side code, such as JavaScript, into generated documents. Therefore, evidence of such an attack might consist of fragments of client-side code contained in parameter values. Because of the insertion of specific HTML tags or attributes as well as the use of characters common in browser scripting languages, this type of attack often results in a violation of the learned structure and character distribution of a parameter.

Consequently, the XSS heuristic is applied to an anomalous attribute value if any of these models are involved in the initial detection step. The heuristics currently used for this class include a set of scans for common syntactic elements of the JavaScript language or HTML fragments (e.g., `<script/>`, DOM event handler attributes, or angle brackets under several encodings).

4.3.3 SQL injection

SQL injection attacks consist of malicious modifications to SQL queries, usually by escaping an input to a query parameter that allows an attacker to execute unauthorized SQL commands. Because of the insertion of these escape characters, SQL injection attacks generally result in the violation of the learned structure of attribute values.

Therefore, the heuristic specific to SQL injection are activated if the structure model detects an anomaly. The heuristic then scans the anomalous value for common SQL language keywords and syntactic elements (e.g., `SELECT`, `INSERT`, `UPDATE`, `DELETE`, `'`, or `--`).

4.3.4 Buffer overflows

Buffer overflow attacks typically involve sending a large amount of data that overflows an allocated buffer, allowing the attacker to overwrite control flow information or application data. Buffer overflow attacks against web applications typically manifest themselves as attribute values that deviate significantly from established profiles of normal behavior. Therefore, the heuristic for inferring the presence of a buffer overflow attack will be activated if any of the character distribution, length, or structure models report an anomaly. The heuristic performs a simple scan over the anomalous value for binary values (i.e., ASCII values greater than 0x7f), which are typical of basic buffer overflow attacks. More sophisticated classification techniques could be substituted, however, with associated tradeoffs in performance [121].

4.4 Evaluation

The set of components responsible for anomaly signature generation, anomaly clustering, and attack class inference were evaluated in terms of false positive rate reduction and attack classification accuracy. All experiments were conducted on a Pentium IV 1.8 GHz machine with 1 GB of RDRAM.

4.4.1 False positive rate reduction

In order to evaluate the false positive rate of WEBANOMALY, data sets from two universities, TU Vienna and UC Santa Barbara, were analyzed by the system. To this end, a client was written to replay the requests to a honeypot web server while a misuse detection system monitored the network link between the client and server. All requests corresponding to attacks detected by the misuse detector were removed from the data sets. Also, since many of the attacks were intended for Microsoft IIS while the data sets corresponded to Apache web servers, many attacks were removed simply by removing requests for non-existent documents identified by the `404` HTTP return code.

WEBANOMALY was configured with an initially empty anomaly signature set, and default learning, detection, and similarity threshold were used. The learning phase was performed over the first 1,000 examples of a specific web application attribute, at which point the attached profile switched into the detection phase. During this phase, any alerts reported by the system were flagged as false positives, under the assumption that the data set was free of attacks. The results of the experiment are shown in Table 4.1.[3]

[3] We note that the fixed training length used in this experiment is an artifact due to the use of an earlier

Data set	Queries	False positives	FPR	Clusters	Cluster FPR
TU Vienna	737,626	14	1.90×10^{-5}	2	3.00×10^{-6}
UCSB	35,261	513	1.45×10^{-2}	3	8.50×10^{-5}

Table 4.1: False positive results for both raw and clustered anomalies.

During analysis of the TU Vienna data set, the detection system produced 14 alerts over 737,626 queries, resulting in a low baseline false positive rate. We believe this attests to the ability of the anomaly detection models to accurately capture the "normal" behavior of attribute values during the learning phase. The addition of the anomaly clustering components, however, improved this even further by allowing the system to collapse 14 alerts into 2 clusters. When these clusters were examined, we found that each of the groups indeed represented related alerts. For the first group, an IMAP mailbox was repeatedly accessed through the imp webmail application, which had not been observed during the training phase. In response, the token finder generated an alert, and the resulting anomaly signature allowed the system to cluster the alerts together accordingly.[4] For the second cluster, developers of a custom web application specified invalid values to an attribute during test invocations of their program. In this case, the attribute length model detected an anomaly, and the resulting anomaly signature correctly grouped subsequent variations on the input errors with the first instance.

The results of the clustering components during analysis of the UC Santa Barbara data set were even more dramatic. The detection system reported 513 alerts over 35,261 queries, resulting in a false positive rate several orders of magnitude greater than the

version of WEBANOMALY. Additionally, we note that the baseline false positive rate for the system has improved considerably since this experiment was conducted.

[4] Incidentally, this would be a reasonable case to retrain the associated models, in order to incorporate the characterstics of the legitimate value into the attribute profile.

TU Vienna data set. Due to our anomaly clustering technique, however, the 513 alerts were partitioned into 3 clusters. Manual inspection of the aggregated alerts demonstrated that, as in the case of the TU Vienna data set, the groups were comprised of related alerts. The first cluster was composed of a series of anomalous queries to the whois.pl user lookup script, which expects a `name` attribute with a valid username as the value. In this case, the grouped alerts all possessed the `name` attribute value `teacher+assistant++advisor`, possibly as the result of a bad hyperlink reference to the script from elsewhere on the department website. In this case, the character distribution model detected an anomalous number of "a" characters. The second cluster was identical in nature to the first, except that the `name` argument value was `dean+of+computer+science`. For this cluster, the character distribution detected an anomalous number of "e" characters. The final cluster was composed of several alerts on an optional argument to the whois.pl script named `showphone`, which takes either a `yes` or `no` value as an argument. In this case, the alerts were attributed to an uppercase `YES` value, which the token finder correctly identified as anomalous.

4.4.2 Attack inference

To evaluate the effectiveness of the attack inference heuristics, a number of attacks comprising the different attack classes that the heuristics are intended to classify were injected into the TU Vienna data set. This data set was chosen because legitimate invocations of the vulnerable applications were previously present in the data. Ten variations of each distinct attack were injected throughout the data set, utilizing mutation techniques from the Sploit framework [124]. WEBANOMALY was configured with the

Attack	Alerting Models	Classification	Accuracy
csSearch	Integer, Char. Distribution	XSS	100%
htmlscript	Integer, Structure	Directory traversal	100%
imp	Integer, Char. Distribution	XSS	100%
phorum	Token, Integer, Char. Distribution	Buffer overflow	100%
phpnuke	Integer, Structure	SQL injection	100%
webwho	Integer	None	100%

Table 4.2: Attack classification results.

exact set of parameters as in the previous experiment. The results of the experiment are shown in Table 4.2.

From the experimental results, we first note that all instances of each attack were determined to be anomalous by WEBANOMALY. Additionally, in each case, all instances of a given attack were classified into a single cluster. Finally, each of the clusters were correctly characterized by the attack inference heuristics. The only attack that was not classified as belonging to a known class of attacks was the webwho attack. This, however, we consider to be correct behavior, as the webwho attack exploited an application-specific input validation error for which the system includes no heuristics. It is important to note, however, that the anomaly was still detected, and further variations were clustered correctly. Indeed, although a variety of models provided the initial decision that the request was anomalous, in each case the anomaly signature generation procedure was able to match subsequent variations of the same attack. We believe that this demonstrates the power of our technique with respect to its ability to group similar anomalies.

Data set	Requests	Request rate	Analysis Time	Analysis Rate
TU Vienna	737,626	0.107095 req/s	934 s	788.06 req/s
UCSB	35,261	0.001360 req/s	64 s	550.95 req/s

Table 4.3: Detection performance results.

4.4.3 Performance

The performance of the detection system was evaluated in terms of both CPU time and memory usage when run on both attack-free data sets from TU Vienna and UC Santa Barbara. Both metrics are important for the real-world applicability of WEBANOMALY, since, in the ideal case, it would be run in real-time on hardware available to most website operators. The same parameters used from the previous experiments were used for this experiment. Ten runs were performed for each data set, and the elapsed times were averaged. The results of the time required for analysis by the system are displayed in Table 4.3.

For both data sets, WEBANOMALY was able to maintain a processing rate orders of magnitude above the rate of requests processed by the respective web servers. For instance, in the case of the TU Vienna data set, the request analysis was performed approximately 7,000 times as quickly as actual requests were processed. From this, we conclude that for many sites, the detection system is capable of performing its analysis in real-time.

In addition to CPU usage, an analysis of the memory utilization of the system was performed. The results of this evaluation showed that WEBANOMALY did not require substantial memory resources once the profiles were established.

4.5 Conclusions

This chapter has presented an approach to addressing two limitations of anomaly-based intrusion detection systems. First, *anomaly signatures* are used to cluster similar anomalies that are likely to represent groups of either true positives or true negatives. This reduces the burden on security administrators in dealing with the high volumes of false positives that anomaly detectors can potentially produce. Secondly, *attack class inference* techniques are used to label anomaly clusters, allowing anomaly detectors to provide additional information as to the type of attacks that anomalies represent. Taken in combination, these techniques significantly increase the effectiveness of anomaly detection systems in practice, by reducing the negative effects of false positives as well as providing guidance in terms of vulnerability mitigation.

Though false positive reduction and the lack of explanatory power are important issues that must be addressed, more challenges to the successful deployment of anomaly detectors remain. The next chapter discusses an approach to dealing with one such challenge: the scarcity of training data.

Chapter 5

Addressing Training Data Scarcity

The models used by anomaly detection systems to characterize normal behavior are central to the accuracy of the system. Approaches to manually specifying anomaly models do exist; this, however, is a tedious, labor-intensive, and error-prone process. Therefore, most research has instead focused on applying machine learning techniques to automatically derive models of normal behavior from unlabeled training data. Anomaly detectors that incorporate such learning techniques obviate the tedious and error-prone task of creating specifications, and, additionally, are able to adapt to the particular characteristics of the local environment.

In an ideal case, a learning-based web anomaly detection system is deployed in front of one or more web applications hosted at a site, and, in a completely automated fashion, the IDS learns the normal interaction between users and the applications. Once enough training data has been analyzed and the profiles for the protected applications have been

established, the system switches to the detection phase, and it is able to detect attacks that represent anomalies with respect to normal application usage; WEBANOMALY exemplifies this approach.

Learning-based anomaly detection provides the significant benefit of requiring only a modest initial configuration effort to provide effective custom attack detection. This feature is extremely attractive to security administrators, who have neither the resources nor the skills to manually analyze legacy web applications composed of hundreds of resources. Because of this, several commercial web application firewalls incorporate limited forms of learning-based anomaly detection to supplement more traditional misuse-based techniques [15, 31, 11].

In addition to the problem of false positives that has been discussed in Chapter 4, another limitation of anomaly detection that is well-known in the research community is the difficulty of obtaining high-quality training data. Learning-based anomaly detectors critically rely upon the quality of the training data used to construct their models. One requirement of training data sets is that they must be free from attacks. Otherwise, the resulting models will be prone to false negatives, as attack manifestations will have been learned as normal. To address this, several works propose approaches to sanitizing training data sets [29, 20, 58].

Another well-known limitation, and one that can be seen as the dual of the attack-free requirement, is that training data sets should *completely capture the normal behavior* of the protected resources. Unfortunately, to our knowledge, no proposals exist that satisfactorily address the problem. In particular, our experience with the deployment of web application anomaly detectors in real-world environments suggests that the number of

web application component invocations is non-uniformly distributed, where relatively few components dominate, and the remaining components are accessed relatively infrequently. Thus, for those components, it is often impossible to gather enough training data to accurately model their normal behavior. We informally refer to the components that receive insufficient accesses as the "long tail" of a web application.[1] In summary, components that are infrequently accessed lead to undertrained models – that is, models that do not accurately characterize normal behavior accurately.

This chapter addresses the problem of undertrained models by exploiting natural similarities among all web applications. The key observation is that the values of the parameters extracted from HTTP requests can generally be categorized according to their type, such as an integer, date, or string. Moreover, our experiments demonstrate that parameters of similar type induce similar models of normal behavior. Taken together, these results can be leveraged to supplement a lack of training data for one web application component with similar data from another component that has received more requests.

5.1 Training data scarcity

To introduce the problem of training data scarcity, we will refer to the abstract model of a web application initially discussed in Section 3.1 and shown in Figure 3.1. Readers familiar with this material can skim until Section 5.1.1.

[1] Note that we refer to the long tail in reference to a Pareto distribution, and that this does not necessarily imply a power law distribution of these accesses.

$$R_i = \left\{ \begin{array}{l} r_{i,1} = /\texttt{index}, \\ r_{i,2} = /\texttt{article}, \\ r_{i,3} = /\texttt{comments}, \\ r_{i,4} = /\texttt{comments/edit}, \\ r_{i,5} = /\texttt{account/login}, \\ r_{i,6} = /\texttt{account/index}, \\ r_{i,7} = /\texttt{account/password} \end{array} \right\}$$

Figure 5.1: Resources comprising an example web application `blog.example.com`.

Recall that a set of web applications A can be decomposed into a set of *resource paths*, or *components*, R and named *parameters* P. A web application receives sequences of requests $Q = \{q_1, q_2, \ldots\}$ issued by web clients, where each query can be represented by the tuple $q_i = \langle a_i, r_{i,j}, P_q \rangle$. Here, a_i is a web application, $r_{i,j}$ is a resource path associated with the web application, and $P_q \subseteq P_{i,j}$ is a subset of possible parameter name-value pairs that $r_{i,j}$ accepts. The web application generates a sequence of responses to queries $S = \{s_1, s_2, \ldots\}$, where a one-to-one mapping exists between each pair (q_i, s_i). Each response can be represented by the tuple $s_i = \langle K_q, d_q \rangle$, where K_q is a set of cookies to be instantiated or cleared on the client, and d_q is a document (e.g., HTML, JSON) to be interpreted by the client.

A web application $a_i = \texttt{blog.example.com}$ might be composed of the resources shown in Figure 5.1. In this example, resource path $r_{i,7}$ might take a set of parameters as part of the HTTP request such as $P_{i,7} = \{p_{i,7,1} = \texttt{oldpw}, p_{i,7,2} = \texttt{newpw}\}$. A client might

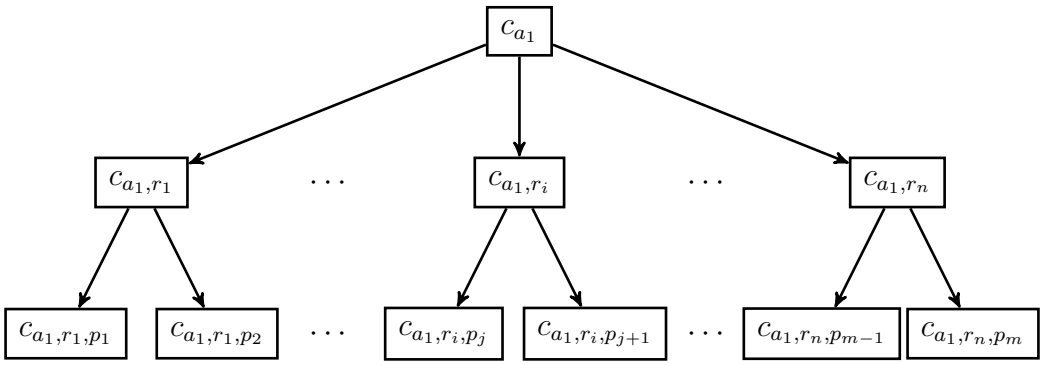


Figure 5.2: The hierarchy of models constructed by WEBANOMALY. Session profiles are created across the entire web application a_1 at the root of the hierarchy. Document profiles are created for each unique resource r_i. Parameter profiles are created for each unique resource and parameter (r_i, p_j).

issue a query to $r_{i,7}$ of the form

$$q = \left\{ \begin{array}{r} \texttt{blog.example.com}, \\ \texttt{/account/password}, \\ \{(\texttt{oldpwd}, \texttt{foo}), (\texttt{newpw}, \texttt{bar})\} \end{array} \right\}.$$

The web application might issue a response of the form $s = \{\emptyset, \text{“}\texttt{<html>}\ldots\text{”}\}$, indicating that no cookies are to be set, and an HTML document is to be rendered.

During the initial *training phase*, the anomaly detection system learns the behavior of the monitored web applications in terms of models. The hierarchy of models created by WEBANOMALY is presented in Figure 5.2, and closely mirrors the abstract model of web applications. At each node of the abstract model, a set of related models is instantiated to model various features of that node. These model sets are known as

profiles, and are represented by a tuple of models

$$c = \left\langle m_1^{(\cdot)}, m_2^{(\cdot)}, \ldots, m_n^{(\cdot)} \right\rangle .$$

Specifically, for each parameter $p_{i,j,k}$, a parameter profile is created that can be represented by the tuple

$$c_{p_{i,j,k}} = \left\langle m^{(\text{tok})}, m^{(\text{int})}, m^{(\text{len})}, m^{(\text{char})}, m^{(\text{struct})} \right\rangle ,$$

where $m^{(\text{tok})}$ is the token model, $m^{(\text{int})}$ is the integer model, $m^{(\text{len})}$ is the string length model, $m^{(\text{char})}$ is the character distribution model, and $m^{(\text{struct})}$ is the structure model.

For sequences of HTTP requests $Q = \{q_1, q_2, \ldots\}$ from a given client, the profile

$$c_{a_i} = \left\langle m^{(\text{int})}, m^{(\text{sess})} \right\rangle$$

is applied, where $m^{(\text{sess})}$ is the session model.

Finally, the structure of responses $r_{i,j}$ generated by the monitored web applications is characterized by the profile

$$c_{r_{i,j}} = \left\langle m^{(\text{doc})} \right\rangle ,$$

where $m^{(\text{doc})}$ is the document model.

$m^{(\text{tok})}$ models parameter values as a small, finite set of legal tokens. $m^{(\text{int})}$ and $m^{(\text{len})}$ describe legitimate distributions for integers and string lengths, respectively, using the

non-parametric Chebyshev inequality. $m^{(\text{char})}$ models character strings as a ranked frequency histogram, or *Idealized Character Distribution* (ICD), that is compared using the χ^2 test. $m^{(\text{struct})}$ models sets of character strings by inducing a Hidden Markov Model (HMM). The HMM encodes a probabilistic grammar that can produce a superset of strings observed in a training data set. $m^{(\text{sess})}$ adapts a combination of the integer model and structure model to capture the normal sequences of request invocations expected by the modeled web application, as well as the inter-arrival times of the individual queries. Finally, $m^{(\text{doc})}$ models the normal structures of documents generated by the monitored web applications, as well as the incidence and positions of sensitive data and client-side code within those documents. For more details on the models and how they are constructed, please refer to Section 3.3.

After converging to a fixed point, individual models switch to the *detection phase*. Here, observed values are compared to profiles to determine how well each value fits the learned models. Bayesian networks are then used to compute an aggregated anomaly score on the interval $[0, 1]$. If the probability of a value is less than a certain threshold, an anomaly is generated.

5.1.1 The problem of non-uniform training data

Because anomaly detection systems dynamically learn specifications of normal behavior from training data, it is clear that the quality of the detection results critically relies upon the quality of the training data. As mentioned previously, one requirement typically imposed upon training data is that it should be *attack-free*; that is, it should not

contain traces of malicious activity that would induce the resulting models to consider malicious behavior as normal. A number of solutions have been proposed to address this issue; for instance, the use of ensemble learning techniques [20]. Another requirement that training should satisfy is that it should accurately represent the normal behavior of the modeled features. In a sense, this requirement is the dual of the previous one: training data should completely contain all relevant aspects of normal behavior.

The difficulty of obtaining sufficient amounts of training data to accurately model web applications is intuitively clear. We are, however, not aware of any solutions that can address this issue when sufficient training data is not available. Typically, anomaly-based detectors cannot assume the presence of a testing framework that can be leveraged to generate realistic training data that exercises the web application in a safe, attack-free environment. Instead, the anomaly detection system is deployed in front of live web applications with no *a priori* knowledge of the applications' components and their behavior.

In the case of low-traffic web applications, problems arise if the rate of client requests is inadequate to allow models to train in a timely manner. However, even in the case of high-traffic web applications, a large subset of resource paths can fail to receive enough requests to adequately train the associated models. This phenomenon, which is frequently observed in real-world data sets, is a direct consequence of the fact that requests issued by web clients often follow a non-uniform distribution. To illustrate this point, Figure 5.3 plots the normalized cumulative distribution function of web client resource path invocations for a variety of real-world, high-traffic web applications.[2]

[2]Details on the source of this data are provided in Section 5.3.

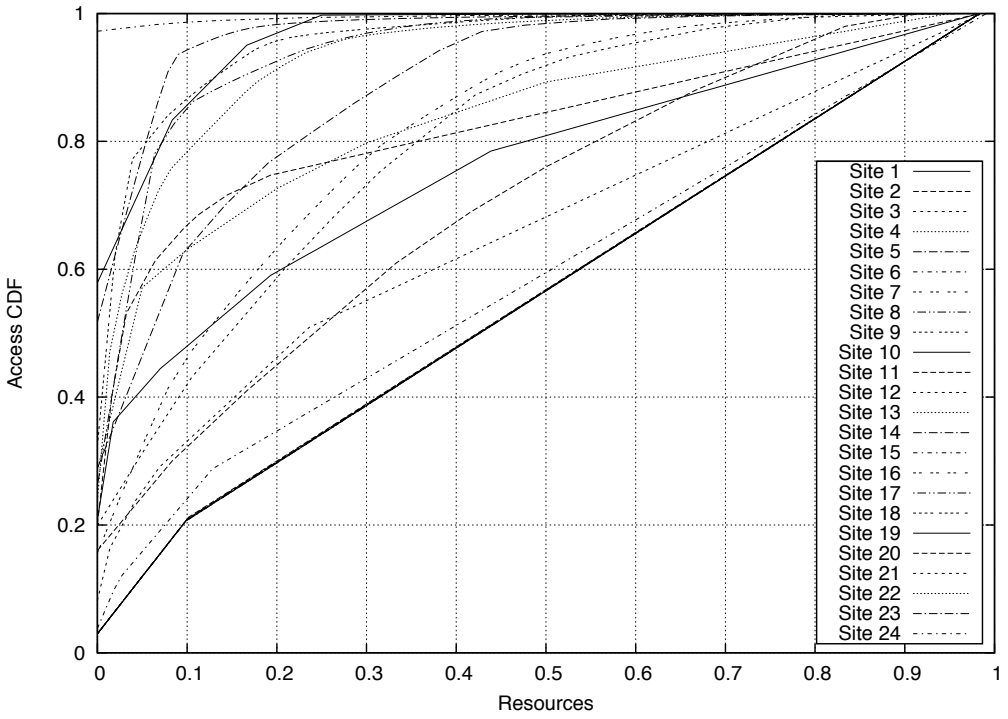


Figure 5.3: Web client resource path invocation distributions from a selection of real-world web applications.

Although several applications have an approximately uniform client access distribution, a clear majority exhibits highly skewed distributions. Indeed, in many cases, a large percentage of resource paths receive a comparatively minuscule number of requests. Returning to the example resources shown in Figure 5.1, assuming an overall request volume of 500,000 requests per day, the example resource path set might result in the client access distribution shown in Figure 5.4.

Clearly, profiles for parameters to resource paths such as `/article` will likely receive sufficient training data. This is not true, however, for profiles associated with paths such as `/account/password`. Further exacerbating the situation is the fact that a client request does not necessarily include all possible parameters.

$$
\begin{aligned}
/\texttt{article} &= 475,000 \\
/\texttt{comments} &= 15,000 \\
/\texttt{comments}/\texttt{edit} &= 9,000 \\
/\texttt{account} &= 900 \\
/\texttt{account}/\texttt{password} &= 100
\end{aligned}
$$

Figure 5.4: Example non-uniform web application request distribution.

The unfeasibility of an anomaly detection system to accurately model a large subset of a web application is problematic in itself. We argue, however, that the impact of the problem is magnified by the fact that components of a web application that are infrequently exercised are also likely to contain a disproportionately large share of security vulnerabilities. This is a consequence of the reduced amount of testing that developers invariably perform on less prominent components of a web application, resulting in a higher rate of software defects. In addition, the relatively low request rate from users of the web application results in a reduced exposure rate for these defects. Finally, when flaws are exposed and reported, correcting the flaws may be given a lower priority than those in higher traffic components of a web application.

Therefore, we conclude that a mechanism to address the problem of model undertraining caused by the non-uniform distribution of training data is necessary for a web application anomaly detection system to provide an acceptable level of security.

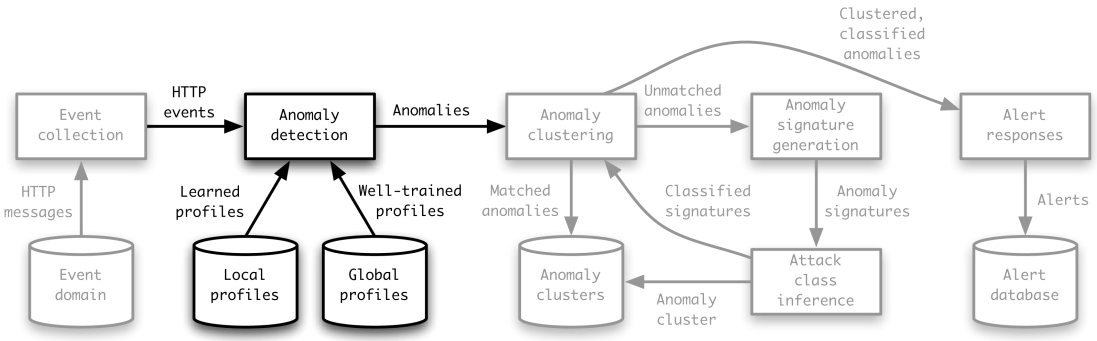


Figure 5.5: Architectural overview of WEBANOMALY, with global profile components highlighted.

5.2 Exploiting global knowledge

The lack of available training data is a fundamental obstacle when constructing accurate profiles for many features of a web application. Without a minimum number of requests to, for instance, a given parameter, it is unfeasible to construct models that encode a reasonably precise approximation of that parameter's normal behavior.

We observe, however, that parameters associated with the invocation of components belonging to different web applications often exhibit a marked similarity to each other. Referring again to the example shown in Figure 5.1, many web applications take an integer value as a unique identifier for a class of objects such as a blog article or comment, as in the case of the `id` parameter. Many web applications also accept date ranges similar to the `date` parameter as identifiers or as constraints upon a search request. Similarly, as in the case of the `title` parameter, web applications often expect a short phrase of text as an input, or perhaps a longer block of text in the form of a comment body. One can consider each of these groupings of similar parameters as distinct

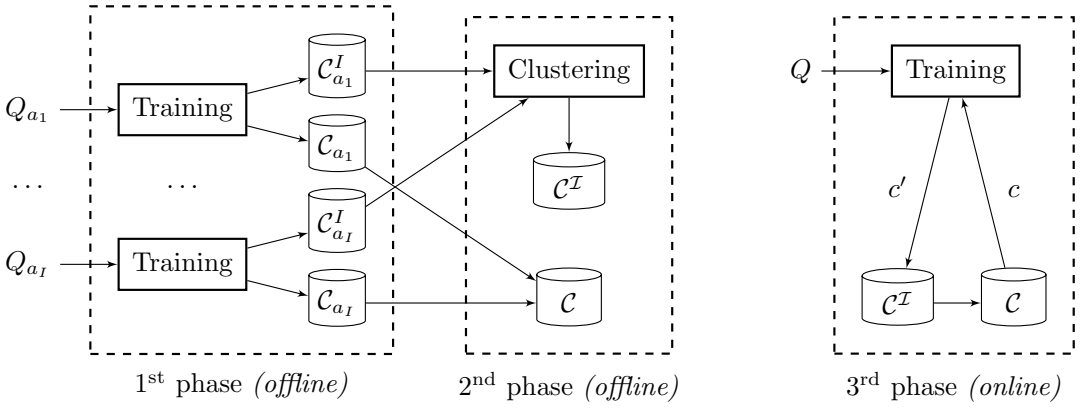


Figure 5.6: Overall procedure. Profiles, both undertrained and well-trained, are collected from a set of web applications. These profiles are processed offline to generate the global knowledge base $\mathcal{C}$ and index $\mathcal{C}^{\mathcal{I}}$. At another web application, given an undertrained profile c', $\mathcal{C}$ can then be queried to find a suitable replacement c.

parameter types.[3]

The key insight behind our approach is that parameters of the same type tend to induce model compositions that are similar to each other in many respects. Consequently, if the lack of training data for a subset of the components of a web application prevents an anomaly detection system from constructing accurate profiles for the parameters of those components, it is possible to substitute profiles for similar parameters of the same type that were learned when enough training data was available. In Section 5.3, the experiments we conducted on real-world data demonstrate that the aforementioned insight is valid.

Our approach is composed of three phases; a graphical overview of the process is presented in Figure 5.6. The *first phase* is an extension of the training procedure originally

[3]This need not necessarily correspond to the concept of types as understood in the programming languages context.

implemented in [66], where undertrained versions of profiles are recorded in addition to their final states. In the *second phase*, a global knowledge base of profiles $\mathcal{C} = \bigcup_{a_i} \mathcal{C}_{a_i}$ is constructed offline, where $\mathcal{C}_{a_i}$ are knowledge bases containing only well-trained, stable profiles from anomaly detection systems previously deployed on a set of web applications $\bigcup_i a_i$. A knowledge base $\mathcal{C}^{\mathcal{I}} = \bigcup_{a_i} \mathcal{C}^{\mathcal{I}}_{a_i}$ of undertrained profiles is then constructed as an index into $\mathcal{C}$, where $\mathcal{C}^{\mathcal{I}}_{a_i}$ is a knowledge base of undertrained profiles from the web application a_i. Additionally, we define a mapping $f : \left\{\mathcal{C}^{\mathcal{I}}\right\} \times \mathcal{C}_{a_i} \mapsto \mathcal{C}$ between undertrained and well-trained profiles.

The *third phase* is performed online. For any new web application where insufficient training data is available for a component's parameter, the anomaly detector first extracts the undertrained profile c'. Then, the global knowledge base $\mathcal{C}$ is queried to find a similar, previously constructed profile $f\left(\mathcal{C}^{\mathcal{I}}, c'\right) = c$. The well-trained profile c is then substituted for the undertrained profile c' in the detection process.

5.2.1 Phase I: Creating undertrained models

In the first phase, the model training algorithm described in Section 3.3 was modified to generate undertrained profiles in addition to normal profiles from a data set. These undertrained profiles are generated using the following procedure. Let

$$Q^{(p)}_{a_i} = \left\{q^{(p)}_1, q^{(p)}_2, \ldots\right\}$$

denote a sequence of client requests containing parameter p for a given web application. Over $Q^{(p)}_{a_i}$, profiles are deliberately undertrained on randomly sampled κ-sequences,

where κ can take values from $\bigcup_{i=0}^{8} 2^i$.[4] Each of the resulting profiles is then added to a knowledge base $\mathcal{C}_{a_i}^{\mathcal{I}}$.

In general, κ corresponds to a number of training samples that is considered insufficient to accurately characterize a feature. As discussed in Section 3.3, however, a criterion based on the notion of *model stability* is used to determine the length of a training phase. Let $\kappa_{\text{stable}}^{m^{(\cdot)}}$ denote the number of samples required for a model to converge to a fixed point – that is, to stabilize. Then, we consider κ to be such that $\kappa \ll \kappa_{\text{stable}}^{m^{(\cdot)}}$. Furthermore, we denote the number of training samples required for a profile to converge as

$$\kappa_{\text{stable}}^{c} = \max_{m^{(\cdot)}} \kappa_{\text{stable}}^{m^{(\cdot)}}. \tag{5.2.1}$$

At the end of this phase, the final state of each well-trained, or stable, profile is stored in a knowledge base $\mathcal{C}_{a_i}$. Both $\mathcal{C}_{a_i}$ and $\mathcal{C}_{a_i}^{\mathcal{I}}$ are collected from each web application, and serve as input to the next phase.

5.2.2 Phase II: Building profile knowledge bases

The second phase consists of processing the output of the first phase, namely the sets of knowledge bases of both undertrained and well-trained profiles learned from a variety of web applications. The goal is to create $\mathcal{C}$ and $\mathcal{C}^{\mathcal{I}}$, global knowledge bases of well-trained and undertrained profiles, respectively, and a mapping between the two, allowing $\mathcal{C}^{\mathcal{I}}$ to serve as an index to $\mathcal{C}$.

[4] A discussion of appropriate values for κ is deferred until Section 5.2.4.

Constructing global knowledge base indices

The construction of the undertrained profile database $\mathcal{C}^{\mathcal{I}}$ begins by merging a set of knowledge bases $\left\{\mathcal{C}^{\mathcal{I}}_{a_1}, \mathcal{C}^{\mathcal{I}}_{a_2}, \ldots, \mathcal{C}^{\mathcal{I}}_{a_I}\right\}$ that have previously been built by WEBANOMALY over a set of web applications $\bigcup_i a_i$ during the first phase. The profiles in $\mathcal{C}^{\mathcal{I}}$ are then clustered to group profiles that are semantically similar to each other. Profile clustering is performed in order to time-optimize query execution when using $\mathcal{C}^{\mathcal{I}}$ as an index into $\mathcal{C}$. The resulting clusters of profiles in $\mathcal{C}^{\mathcal{I}}$ are denoted by $H^{\mathcal{I}} = \bigcup_i h^{\mathcal{I}}_i$. In this work, an agglomerative hierarchical clustering algorithm using group average linkage was applied, although the clustering stage is, in principle, agnostic as to the specific algorithm. For an in-depth discussion of clustering algorithms and techniques, we refer the reader to [140].

Central to any clustering algorithm is the distance function, which defines how distances between the objects to be clustered are calculated. A suitable distance function must reflect the semantics of the objects under consideration, and should satisfy two conditions: 1) the overall similarity between elements within the same cluster is maximized, and 2) the similarity between elements within different clusters is minimized.

We define the distance between two profiles to be the sum of the distances between the models comprising each profile. More formally, the distance between the profiles c_i and c_j is defined as

$$(c_i, c_j) = \frac{1}{|c_i \bigcap c_j|} \sum_{m_i^{(u)}, m_j^{(u)} \in c_i \bigcap c_j} \delta_u \left(m_i^{(u)}, m_j^{(u)} \right), \tag{5.2.2}$$

where $\delta_u : \mathcal{M}_u \times \mathcal{M}_u \mapsto [0, 1]$ is the distance function defined between models of type $u \in U = \{\text{tok}, \text{int}, \text{len}, \text{char}, \text{struct}\}$.

The token model $m^{(\text{tok})}$ is represented as a set of unique tokens observed during the training phase. Therefore, two token models $m_i^{(\text{tok})}$ and $m_j^{(\text{tok})}$ are considered similar if they contain similar sets of tokens. Accordingly, the distance function for token models is defined as the Jaccard distance [17]

$$\delta_{\text{tok}}\left(m_i^{(\text{tok})}, m_j^{(\text{tok})}\right) = 1 - \frac{\left|m_i^{(\text{tok})} \bigcap m_j^{(\text{tok})}\right|}{\left|m_i^{(\text{tok})} \bigcup m_j^{(\text{tok})}\right|}. \tag{5.2.3}$$

The integer model $m^{(\text{int})}$ is parameterized by the sample mean μ and variance σ^2 of observed integers. Two integer models $m_i^{(\text{int})}$ and $m_j^{(\text{int})}$ are similar if these parameters are also similar. Consequently, the distance function for integer models is defined as

$$\delta_{\text{int}}\left(m_i^{(\text{int})}, m_j^{(\text{int})}\right) = \frac{\left\|\frac{\sigma_i^2}{\mu_i^2} - \frac{\sigma_j^2}{\mu_j^2}\right\|}{\frac{\sigma_i^2}{\mu_i^2} + \frac{\sigma_j^2}{\mu_j^2}}. \tag{5.2.4}$$

As the length model is internally identical to the integer model, its distance function δ_{len} is defined similarly.

Recall that the character distribution model $m^{(\text{char})}$ learns the frequencies of individual characters comprising strings observed during the training phase. These frequencies are then ranked and coalesced into n bins to create an ICD. Two character distribution models $m_i^{(\text{char})}$ and $m_j^{(\text{char})}$ are considered similar if each model's ICDs are similar.

Therefore, the distance function for character distribution models is defined as

$$\delta_{\text{char}}\left(m_i^{(\text{char})}, m_j^{(\text{char})}\right) = \sum_{l=0}^{n} \frac{\|b_i\,(l) - b_j\,(l)\|}{\max_{k=i,j} b_k\,(l)}, \tag{5.2.5}$$

where $b_i\,(k)$ is the value of bin k for $m_i^{(\text{char})}$.

The structural model $m^{(\text{struct})}$ builds an HMM by observing a sequence of character strings. The resulting HMM encodes a probabilistic grammar that can produce a superset of the strings observed during the training phase. The HMM is specified by the tuple $\langle \mathbb{S}, \mathbb{O}, M_{\mathbb{S}\times\mathbb{S}}, P\,(\mathbb{S}, \mathbb{O})\,, P\,(\mathbb{S})\rangle$. Several distance metrics have been proposed to evaluate the similarity between HMMs [114, 80, 115, 56]. Their time complexity, however, is non-negligible. Therefore, we adopt a less precise, but considerably more efficient, distance metric between two structural models $m_i^{(\text{struct})}$ and $m_j^{(\text{struct})}$ as the Jaccard distance between their respective emission sets

$$\delta_{\text{struct}}\left(m_i^{(\text{struct})}, m_j^{(\text{struct})}\right) = 1 - \frac{|\mathbb{O}_i \bigcap \mathbb{O}_j|}{|\mathbb{O}_i \bigcup \mathbb{O}_j|}. \tag{5.2.6}$$

Constructing a global knowledge base

Once a knowledge base of undertrained models $\mathcal{C}^{\mathcal{I}}$ has been built, the next step is to construct a global knowledge base $\mathcal{C}$. This knowledge base is composed of the individual, well-trained knowledge bases from each web application as recorded during the first phase; that is, $\mathcal{C} = \bigcup_i \mathcal{C}_{a_i}$. Because undertrained profiles are built for each well-trained profile in $\mathcal{C}$, a well-defined mapping $f' : \mathcal{C}^{\mathcal{I}} \mapsto \mathcal{C}$ exists between $\mathcal{C}^{\mathcal{I}}$ and $\mathcal{C}$. Therefore, when a web application parameter is identified as likely to be undertrained,

the corresponding undertrained profile c' can be compared to a similar undertrained profile in $\mathcal{C}^{\mathcal{I}}$, which is then used to select a corresponding stable profile from $\mathcal{C}$.

5.2.3 Phase III: Mapping undertrained profiles

With the construction of a global knowledge base $\mathcal{C}$ and an undertrained knowledge base $\mathcal{C}^{\mathcal{I}}$, we can perform online querying of $\mathcal{C}$. That is, given an undertrained profile c' from an anomaly detector deployed over a web application a_i, the mapping $f : \left\{\mathcal{C}^{\mathcal{I}}\right\} \times \mathcal{C}_{a_i} \mapsto \mathcal{C}$ is defined as follows. A nearest-neighbor match is performed between c' and the previously constructed clusters $H^{\mathcal{I}}$ from $\mathcal{C}^{\mathcal{I}}$ to discover the most similar cluster of undertrained profiles. This is done to avoid a full scan of the entire knowledge base, which would be prohibitively expensive due to the cardinality of $\mathcal{C}^{\mathcal{I}}$.

Then, using the same distance metric defined in (5.2.2), a nearest-neighbor match is performed between c' and the members of $H^{\mathcal{I}}$ to discover the most similar undertrained profile $c^{\mathcal{I}}$. Finally, the global, well-trained profile $f'\left(c^{\mathcal{I}}\right) = c$ is substituted for c' for the web application a_i.

To make explicit how global profiles can be used to address a scarcity of training data, consider the example of Figure 5.4. Since the resource path `/account/password` has received only 100 requests, the profiles for each of its parameters $\{\texttt{id}, \texttt{oldpw}, \texttt{newpw}\}$ are undertrained. In the absence of a global knowledge base, the anomaly detector would provide no protection against attacks manifesting themselves in the values passed to any of these parameters. If, however, a global knowledge base and index are available, the situation is considerably improved. Given $\mathcal{C}$ and $\mathcal{C}^{\mathcal{I}}$, the anomaly detector

can simply apply f to each of the undertrained parameters to find a well-trained profile from the global knowledge base that accurately models a parameter with similar semantics, even when the model is for *another web application*. Then, these profiles can be substituted for the undertrained profiles for each of $\{\texttt{id}, \texttt{oldpw}, \texttt{newpw}\}$. As will be demonstrated in the following section, the substitution of global profiles provides an acceptable detection accuracy for what would otherwise be an unprotected component. Indeed, without a global profile, none of the attacks against that component would be detected.

5.2.4 Mapping quality

The selection of an appropriate value for κ is central to both the efficiency and the accuracy of querying $\mathcal{C}$. Clearly, it is desirable to minimize κ in order to be able to index into $\mathcal{C}$ as quickly as possible once a parameter has been identified to be subject to undertraining at runtime. On the other hand, setting κ too low is problematic, as Figure 5.7a indicates. For low values of κ, profiles are distributed with relatively high uniformity within $\mathcal{C}^{\mathcal{I}}$, such that clusters in $\mathcal{C}^{\mathcal{I}}$ are significantly different than clusters of well-trained profiles in $\mathcal{C}$. Therefore, slight differences in the state of the individual models can cause profiles that are close in $\mathcal{C}^{\mathcal{I}}$ to map to radically different profiles in $\mathcal{C}$. As $\kappa \rightarrow \kappa_{\text{stable}}$, however, profiles tend to form semantically-meaningful clusters, and tend to approximate those found in $\mathcal{C}$. Therefore, as κ increases, profiles that are close in $\mathcal{C}^{\mathcal{I}}$ become close in $\mathcal{C}$ under f – in other words, f becomes *robust* with respect to model semantics.[5]

[5]Our use of the term "robustness" is related, but not necessarily equivalent, to the definition of robustness in statistics (i.e., the property of a model to perform well even in the presence of small changes

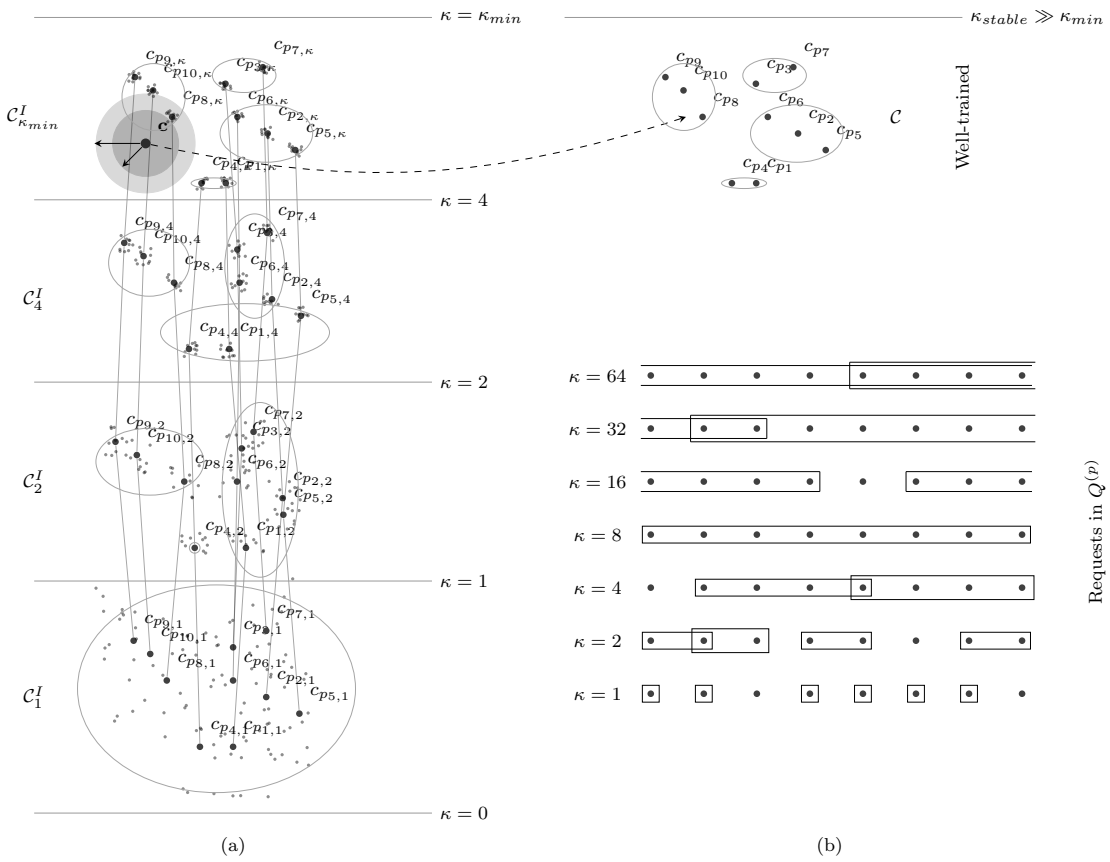


Figure 5.7: Procedure for building global knowledge base indices (a) by sub-sampling (b) the training set Q.

A principled criterion is required for balancing quick indexing against a robust profile mapping. Accordingly, we first construct a candidate knowledge base $\mathcal{C}_\kappa^\mathcal{I}$ for a given κ. Additionally, we cluster the profiles in $\mathcal{C}$ as in the case of the undertrained knowledge base. Then, we define a *robustness metric* as follows. Recall that $H^\mathcal{I} = \bigcup_i h_i^\mathcal{I}$ is the set of clusters in $\mathcal{C}^\mathcal{I}$, and let $H = \bigcup_i h_i$ be the set of clusters in $\mathcal{C}$. Let $g : H^\mathcal{I} \mapsto \mathbb{Z}^+$ be a mapping from an undertrained cluster to the maximum number of elements in that cluster that map to the same cluster in $\mathcal{C}$. The robustness metric ρ is then defined as

$$\rho\left(\mathcal{C}^\mathcal{I}\right) = \frac{1}{|\mathcal{C}^\mathcal{I}|} \sum_i g\left(h_i^\mathcal{I}\right). \tag{5.2.7}$$

With this metric, an appropriate value for κ can now be chosen as

$$\kappa_{\text{min}} = \min_\kappa \left(\rho\left(\mathcal{C}_\kappa^\mathcal{I}\right) \geq \rho_{\text{min}}\right), \tag{5.2.8}$$

where ρ_{min} is a minimum robustness threshold.

5.3 Evaluation

The goal of this evaluation is threefold. First, we investigate the effects of profile clustering, and support the notion of parameter types by examining global knowledge base clusters. Then, we study how the quality of the mapping between undertrained profiles and well-trained profiles improves as the training slice length κ is increased. Finally, we present results regarding the accuracy of a web application anomaly detection sys-

in the underlying assumptions.)

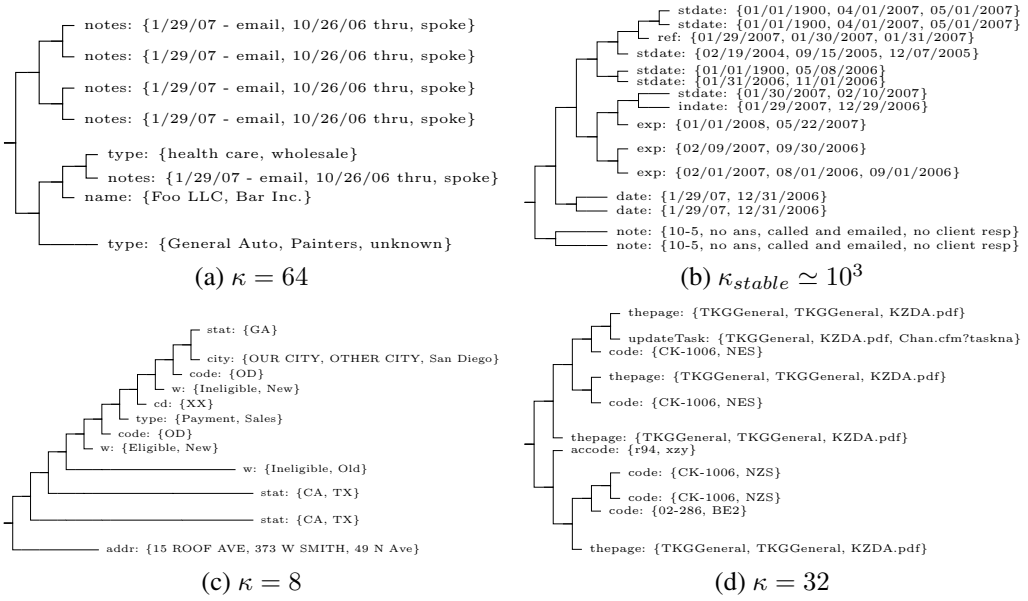
tem incorporating the application of a global knowledge base to address training data scarcity.

The experiments that follow were conducted using a data set drawn from real-world web applications deployed on both academic and industry web servers. Examples of representative applications include payroll processors, client management, and online commerce sites. For each application, the full content of each HTTP connection observed over a period of approximately three months was recorded. The resulting flows were then filtered using snort to remove known attacks. In total, the data set contains 823 distinct web applications, 36,392 unique components, 16,671 unique parameters, and 58,734,624 HTTP requests.[6]

5.3.1 Profile clustering quality

To evaluate the accuracy of the clustering phase, we first built a global knowledge base $\mathcal{C}$ from a collection of well-trained profiles using the procedure described in Section 5.2.1. The profiles were trained on a subset of the aforementioned data, containing traffic involving a wide variety of web applications. This subset was composed of 603 web applications, 27,990 unique resource paths, 9,023 unique parameters, and 3,444,092 HTTP requests. The clustering algorithm described in Section 5.2.2 was then applied to group profiles according to their parameter type. Sample results from this clustering are shown in Figure 5.8b. Each leaf node corresponds to a profile and displays the parameter name and a few representative sample values corresponding to

[6]Unfortunately, due to contractual agreements, we are unable to disclose specific information identifying the web applications themselves.



(a) $\kappa = 64$ (b) $\kappa_{stable} \simeq 10^3$

(c) $\kappa = 8$ (d) $\kappa = 32$

Figure 5.8: Dendrograms of the clustering of $\mathcal{C}$, (a-b), and $\mathcal{C}^{\mathcal{I}}$, (c-d), at various κ. Each leaf represents a profile and includes the name of the parameter and samples values observed during training. As κ increases, profiles are clustered more accurately.

the parameter.

As the partial dendrogram indicates, the resulting clusters in $\mathcal{C}$ are accurately clustered by parameter type. For instance, date parameters from different web applications are grouped into a single hierarchy, while unstructured text strings are grouped into a separate hierarchy.

The following experiment investigates how κ affects the quality of the final clustering.

5.3.2 Profile mapping robustness

Recall that in order to balance the robustness of the mapping f between undertrained profiles and global profiles against the speed with which undertraining can be addressed, it is necessary to select an appropriate value for κ. To this end, we generated undertrained knowledge bases for increasing values of $\kappa = \{1, 2, 4, 8, 16, 32, 64\}$ from the same data set used to generate $\mathcal{C}$, following the procedure outlined in Section 5.2.2. Partial dendrograms for various κ are presented in Figure 5.8.

At low values of κ – for example, Figure 5.8c – the clustering process exhibits non-negligible systemic errors. For instance, the parameter `stat` should be clustered as a token set of states, but instead is grouped with unstructured strings such as cities and addresses. A more accurate clustering would have dissociated the token and string profiles into well-separated sub-hierarchies.

As shown in Figure 5.8d, larger values of κ lead to more semantically-meaningful groupings. Some inaccuracies are still noticeable, but the clustering process of the sub-hierarchy is significantly better than the one obtained at $\kappa = 8$. A further improvement in the clusters is shown in Figure 5.8a. At $\kappa = 64$, the separation between dates and unstructured strings is sharper; except for one outlier, the two types are recognized as similar and grouped together in the early stages of the clustering process.

Figure 5.9 plots the profile mapping robustness ρ against κ for different cuts of the dendrogram, indicated by D_{max}. D_{max} is a threshold representing the maximum distance between two clusters. For low D_{max}, the "cut" will generate many clusters with a few elements; on the other hand, for high values of D_{max}, the algorithm will tend to form

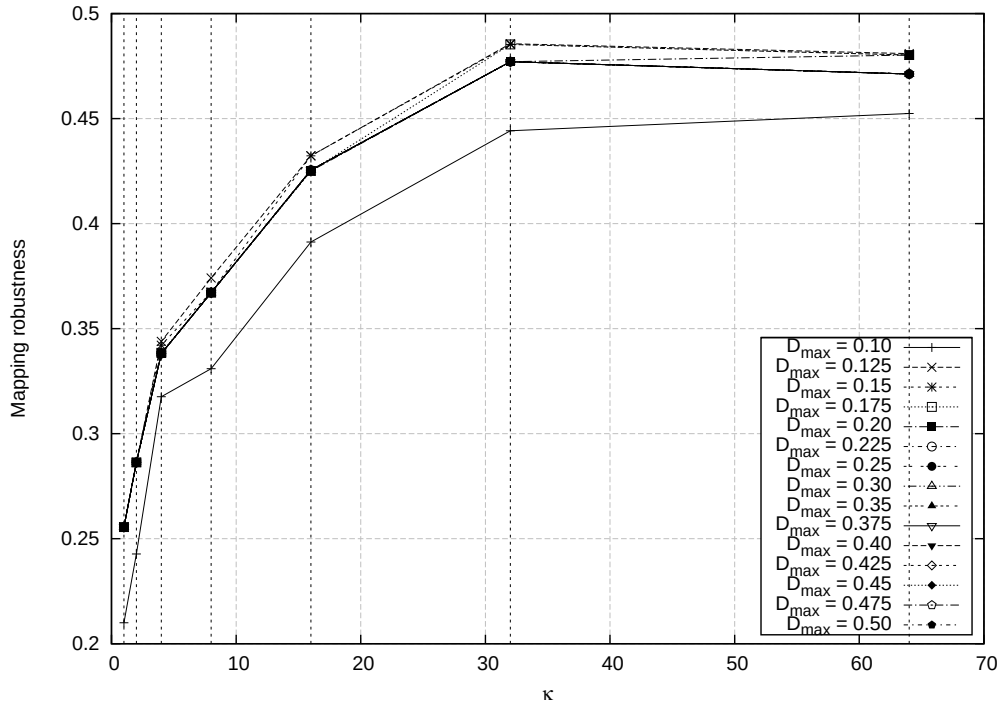


Figure 5.9: Plot of profile mapping robustness for varying κ.

less clusters, each having a larger number of elements.

Figure 5.9 shows two important properties of our technique. First, it demonstrates that the robustness is fairly insensitive to $D_{\max}$. Second, the robustness of the mapping increases with κ until saturation at $32 \leq \kappa \leq 64$. This not only confirms the soundness of the mapping function, but it also provides insights on the appropriate choice of κ_{min} to minimize the delay to global profile lookup while maximizing the robustness of the mapping.

5.3.3 Detection accuracy

Having studied the effects of profile clustering and varying values for κ upon the robustness of the profile mapping f, a separate experiment was conducted in order to evaluate the detection accuracy of a web application anomaly detector incorporating $\mathcal{C}$ and $\mathcal{C}^{\mathcal{I}}$, the knowledge bases constructed in the previous experiments. In particular, the goal of this experiment is to demonstrate that the combination of WEBANOMALY and a global knowledge base exhibits an improved detection accuracy in the presence of training data scarcity.

The data used in this experiment was a subset of the full data set described above, containing traffic from one related set of web applications implementing online commerce sites. This data set was completely disjoint from the one used to construct the global knowledge base and its indices, to prevent any potential for the substitution of profiles from the same application. Additionally, the use of a global knowledge base generated from many types of web applications to address a lack of training data for a specific web application mirrors the intended usage of the technique. In total, this data set consisted of 220 unique real-world web applications, 8,402 unique resource paths, 7,648 distinct parameters, and 55,290,532 HTTP requests.

The threat model that the anomaly detector assumes is that of an attacker attempting to compromise the confidentiality or integrity of data exposed by a web application by injecting malicious code in request parameters.[7] Therefore, a set of 100,000 attacks

[7] Although the anomaly detector used in this study is capable of detecting more complex session-level anomalies, we restrict the threat model to request parameter manipulation because we do not address session profile clustering in this work.

was introduced into the data set. These attacks were real-world examples and variations upon cross-site scripting (XSS), SQL injection, and command execution exploits that manifest themselves in request parameter values.[8] Representative examples of these attacks include:

- malicious JavaScript inclusion

  ```
  <script src="http://example.com/malware.js"></script>;
  ```

- bypassing login authentication

  ```
  ' OR 'x'='x'--;
  ```

- command injection

  ```
  ; cat /etc/passwd | mail attacker@gmail.com #.
  ```

To establish a worst-case bound on the detection accuracy of the system, profiles for each observed request parameter were deliberately undertrained to artificially induce a scarcity of training data for *all* parameters. That is, for each value of $\kappa = \bigcup_{i=0}^{6} 2^i$, the anomaly detector prematurely terminated profile training after κ samples, and then used the undertrained profiles to query $\mathcal{C}$. The resulting global profiles were then substituted for the undertrained profiles and evaluated against the rest of the data set. The sensitivity of the system was varied over the interval $[0, 1]$, and the resulting ROC curves for each κ are plotted in Figure 5.10.

As is clearly indicated, low values of κ result in the selection of global profiles that do not accurately model the behavior of the undertrained parameters. As κ increases,

[8] These attacks remain the most common attacks against web applications.

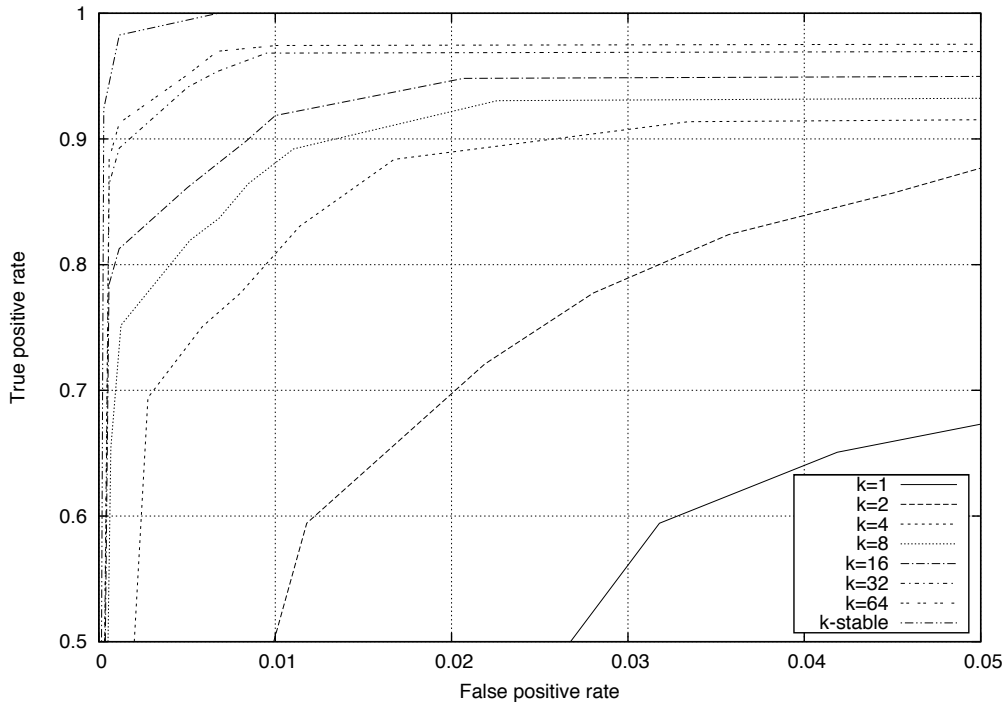


Figure 5.10: Global profile ROC curves for varying κ.

however, the quality of the global profiles returned by the querying process increases as well. In particular, this increase in quality closely follows the mapping robustness plot presented in Figure 5.9. As predicted, setting $\kappa = \{32, 64\}$ leads to fairly accurate global profile selection, with the resulting ROC curves approaching that of fully-trained profiles. This means that even if the component of a web application has received only a few requests, it is possible to achieve effective attack detection by leveraging a global knowledge base. As a consequence, our approach can improve the effectiveness of real-world web application anomaly detection systems.

One concern regarding the substitution of global profiles for local request parameters is that a global profile that was trained on another web application may not detect valid

attacks against the undertrained parameter. Without this technique, however, recall that a learning-based web application anomaly detector would otherwise have no effective model at all, and, therefore, the undertrained parameter would be unprotected by the detection system (i.e., zero true positive rate). Furthermore, the ROC curves demonstrate that while global profiles are in general not as precise as locally-trained models, they do provide a significant level of detection accuracy.[9] Therefore, we conclude that our approach is a useful technique to apply in the presence of undertrained models and, in general, in the case of training data scarcity.

5.4 Conclusions

In this chapter, the predominant case of highly skewed web client access distributions is shown to cause model undertraining. The resulting scarcity of training data is a fundamental challenge to web application anomaly detection that we address through the inclusion of *global knowledge bases*. Global knowledge bases contain well-trained, stable profiles to remediate a local scarcity of training data by exploiting global similarities in web application parameters. We have evaluated the efficacy of this approach over an extensive data set collected from real-world web applications. We found that although using global profiles does result in a small reduction in detection accuracy, the resulting system, when given appropriate parameters, does provide reasonably precise modeling of otherwise unprotected web application parameters.

[9]Note that if global profiles were found to be as accurate as local profiles, this would constitute an argument against site-specific learning of models, since in that case, models could be trained for one web application and applied directly to other web applications.

The next chapter will discuss an approach to dealing with yet another fundamental obstacle to anomaly detection: *concept drift*.

Chapter 6

Addressing Concept Drift

Anomaly-based intrusion detection generally relies upon machine learning techniques to automatically construct models characterizing the normal behavior of certain features. As discussed in the previous chapter, the training data sets used to construct these models must fulfull two requirements. First, the training data must be free from attacks. Second, the training data must completely capture the behavior of the modeled features. Existing work has addressed how training in the presence of noisy data may be accomplished [29, 20, 58], and WEBANOMALY incorporates a set of components specifically designed to address situations involving a scarcity of training data.

One issue that has not been well-studied, however, is the difficulty of adapting to changes in the behavior of the protected applications. By *behavior of a web application*, we refer to the set of features exposed to the end user by an application. In the context of web applications, this corresponds to the set of inputs that are processed and

the outputs that are produced. This issue has become increasingly important because of the nature of web application distribution. In contrast to traditional software distribution mechanisms, web applications are centralized, and therefore easy to update. Therefore, the addition or modification of features and, thus, application behavior over time is common.

Our analysis reveals that significant changes in the behavior of web applications are indeed frequent. We refer to this phenomenon as *web application concept drift*. In the context of anomaly-based detection, this means that legitimate behavior might be misclassified as an attack after an update of the application, causing the generation of false positives. Normally, whenever a new version of an application is deployed in a production environment, a coordinated effort involving application maintainers, deployment administrators, and security experts is required. That is, developers have to inform administrators about the changes that are rolled out, and the administrators have to update or re-train the anomaly models accordingly. Otherwise, the amount of false positives will increase significantly.

We propose a technique that recognizes when anomalous inputs are due to legitimate updates to a web application. In such cases, false positives are suppressed by automatically and selectively re-training the associated models. Moreover, when possible, model parameters can be automatically updated without requiring any re-training. This technique renders the tedious steps described above unnecessary.

This chapter proposes a set of change detection techniques to address the concept drift problem by treating the protected web applications as oracles. We show that HTTP responses contain important information that can be effectively leveraged to update

previously learned models to take changes into account. The results of applying our technique on real-world data show that learning-based anomaly detectors can automatically *adapt to changes*, and, through this technique, are able to reduce their false positive rate without decreasing their detection accuracy.

The following sections introduce a precise notion of concept drift as it relates to web applications. We then motivate the use of our change detection technique by presenting a large-scale study of the prevalence of concept drift in the real world. Then, the design of our system to address concept drift is discussed. Finally, WEBANOMALY with change detection is evaluated to quantify the resulting improvement in detection accuracy.[1]

6.1 Concept drift

To introduce the idea of concept drift, we will refer to the generalized model of web-based applications presented in Section 3.1 and depicted in Figure 3.1. Readers familiar with this material can skim until Section 6.1.2. After describing the problem of concept drift, we show that concept drift is a problem that exists in the real world, and we motivate why it should be addressed. Unless stated differently, we use the shorthand term *anomaly detector* to refer to anomaly-based detectors that leverage unsupervised machine learning techniques.

[1] An earlier version of this work was presentd in [83].

$$R_i = \left\{ \begin{array}{l} r_{i,1} = /\texttt{index}, \\ r_{i,2} = /\texttt{article}, \\ r_{i,3} = /\texttt{comments}, \\ r_{i,4} = /\texttt{comments/edit}, \\ r_{i,5} = /\texttt{account/login}, \\ r_{i,6} = /\texttt{account/index}, \\ r_{i,7} = /\texttt{account/password} \end{array} \right\}$$

Figure 6.1: Resources comprising an example web application `blog.example.com`.

6.1.1 Anomaly detection for web applications

A set of web applications A can be decomposed into a set of *resource paths*, or *components*, R and named *parameters* P. A web application receives sequences of requests $Q = \{q_1, q_2, \ldots\}$ issued by web clients, where each query can be represented by the tuple $q_i = \langle a_i, r_{i,j}, P_q \rangle$. Here, a_i is a web application, $r_{i,j}$ is a resource path associated with the web application, and $P_q \subseteq P_{i,j}$ is a subset of possible parameter name-value pairs that $r_{i,j}$ accepts. The web application generates a sequence of responses to queries $S = \{s_1, s_2, \ldots\}$, where a one-to-one mapping exists between each pair (q_i, s_i). Each response can be represented by the tuple $s_i = \langle K_q, d_q \rangle$, where K_q is a set of cookies to be instantiated or cleared on the client, and d_q is a document (e.g., HTML, JSON) to be interpreted by the client.

A web application $a_i = \texttt{blog.example.com}$ might be composed of the resources shown in Figure 5.1. In this example, resource path $r_{i,7}$ might take a set of parameters as part of the HTTP request such as $P_{i,7} = \{p_{i,7,1} = \texttt{oldpw}, p_{i,7,2} = \texttt{newpw}\}$. A client might

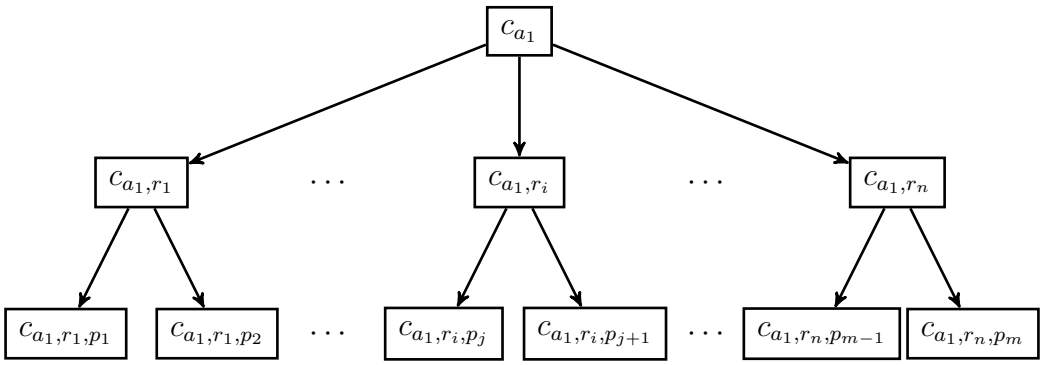


Figure 6.2: The hierarchy of models constructed by WEBANOMALY. Session profiles are created across the entire web application a_1 at the root of the hierarchy. Document profiles are created for each unique resource r_i. Parameter profiles are created for each unique resource and parameter (r_i, p_j).

issue a query to $r_{i,7}$ of the form

$$q = \left\{ \begin{array}{r} \texttt{blog.example.com}, \\ \texttt{/account/password}, \\ \{(\texttt{oldpwd}, \texttt{foo}), (\texttt{newpw}, \texttt{bar})\} \end{array} \right\}.$$

The web application might issue a response of the form $s = \{\emptyset, \text{“}\texttt{<html>}\ldots\text{”}\}$, indicating that no cookies are to be set, and an HTML document is to be rendered.

During the initial *training phase*, the anomaly detection system learns the behavior of the monitored web applications in terms of models. The hierarchy of models created by WEBANOMALY is presented in Figure 6.2, and closely mirrors the abstract model of web applications. At each node of the abstract model, a set of related models is instantiated to model various features of that node. These model sets are known as

profiles, and are represented by a tuple of models

$$c = \left\langle m_1^{(\cdot)}, m_2^{(\cdot)}, \ldots, m_n^{(\cdot)} \right\rangle .$$

Specifically, for each parameter $p_{i,j,k}$, a parameter profile is created that can be represented by the tuple

$$c_{p_{i,j,k}} = \left\langle m^{(\text{tok})}, m^{(\text{int})}, m^{(\text{len})}, m^{(\text{char})}, m^{(\text{struct})} \right\rangle ,$$

where $m^{(\text{tok})}$ is the token model, $m^{(\text{int})}$ is the integer model, $m^{(\text{len})}$ is the string length model, $m^{(\text{char})}$ is the character distribution model, and $m^{(\text{struct})}$ is the structure model.

For sequences of HTTP requests $Q = \{q_1, q_2, \ldots\}$ from a given client, the profile

$$c_{a_i} = \left\langle m^{(\text{int})}, m^{(\text{sess})} \right\rangle$$

is applied, where $m^{(\text{sess})}$ is the session model.

Finally, the structure of responses $r_{i,j}$ generated by the monitored web applications is characterized by the profile

$$c_{r_{i,j}} = \left\langle m^{(\text{doc})} \right\rangle ,$$

where $m^{(\text{doc})}$ is the document model.

$m^{(\text{tok})}$ models parameter values as a small, finite set of legal tokens. $m^{(\text{int})}$ and $m^{(\text{len})}$ describe legitimate distributions for integers and string lengths, respectively, using the

non-parametric Chebyshev inequality. $m^{(\text{char})}$ models character strings as a ranked frequency histogram, or *Idealized Character Distribution* (ICD), that is compared using the χ^2 test. $m^{(\text{struct})}$ models sets of character strings by inducing a Hidden Markov Model (HMM). The HMM encodes a probabilistic grammar that can produce a superset of strings observed in a training data set. $m^{(\text{sess})}$ adapts a combination of the integer model and structure model to capture the normal sequences of request invocations expected by the modeled web application, as well as the inter-arrival times of the individual queries. Finally, $m^{(\text{doc})}$ models the normal structures of documents generated by the monitored web applications, as well as the incidence and positions of sensitive data and client-side code within those documents. For more details on the models and how they are constructed, please refer to Section 3.3.

After converging to a fixed point, individual models switch to the *detection phase*. Here, observed values are compared to profiles to determine how well each value fits the learned models. Bayesian networks are then used to compute an aggregated anomaly score on the interval $[0, 1]$. If the probability of a value is less than a certain threshold, an anomaly is generated.

6.1.2 Web applications are not static

In machine learning, changes in the modeled behavior are known as *concept drift* [109]. Intuitively, the *concept* is the modeled phenomenon – for example, the structure of requests to a web server, or the recurring patterns in the payload of network packets. Thus, variations in the main features of the phenomena under consideration result in

changes, or *drifts*, in the *concept*.

Although the generalization and abstraction capabilities of modern learning-based anomaly detectors are resilient to noise consisting of small, legitimate variations in the modeled behavior, concept drift is difficult to detect and to cope with [64]. The reason is that modeled features may stabilize to different values after a change has occurred. For instance, a string length model could calculate the sample mean and variance of the string lengths that are observed during training. Then, during detection, the Chebyshev inequality is used to detect strings with lengths that significantly deviate from the mean, taking into account the observed variance. Clearly, small differences in the lengths of strings will be considered normal. On the other hand, the mean and variance of the string lengths can completely change because of legitimate and permanent modifications in the web application. In this case, the normal mean and variance will converge to completely different values, resulting in concept drift. If appropriate re-training or manual updates are not performed, the model will classify new, benign strings as anomalous. This can be a human-intensive activity requiring substantial expertise. Therefore, having an automated, black-box mechanism to adjust the parameters is clearly very desirable.

Changes in web applications can manifest themselves in several ways. In the context of learning-based detection of web attacks, those changes can be categorized into three groups: *request* changes, *session* changes, and *response* changes.

Request changes

Changes in requests occur when an application is upgraded to handle different HTTP requests. These changes can be further divided into two groups: *parameter value* changes and *request structure* changes. The former involve modifications of the actual value of the parameters, while the latter occur when parameters are *added*, *removed*, or *renamed*.

Example. A new version of a web forum introduces internationalization (I18N) and localization (L10N). Besides handling different languages, I18N and L10N allow several types of strings to be parsed as valid dates and times. For instance, valid strings for the `datetime` parameter are `'3 May 2009 3:00'`, `'3/12/2009'`, `'3/12/2009 3:00 PM GMT-08'`, `'now'`. In the previous version, valid date-time strings had to conform to the regular expression `'[0-9]{1,2}/[0-9]{2}/[0-9]{4}'`. Even a model with good generalization properties would learn that the field `datetime` is composed of numbers and slashes, with no spaces. Thus, other strings such as `'now'` or `'3/12/2009 3:00 PM GMT-08'` would be flagged as anomalous. Also, in our example, `tz` and `lang` parameters have been added to take into account time zones and languages. To summarize, the new version introduces two classes of changes. Clearly, the parameter domain of `datetime` is modified. Secondly, new parameters are added.

Changes in HTTP requests directly affect the request models. First, parameter value changes affect any models that rely on the parameters' values to extract features. For instance, consider two of the models used in WEBANOMALY: $m^{(\text{char})}$ and $m^{(\text{struct})}$. The former models a string's character distribution by storing the frequency of all the symbols found in the strings during training, while the latter models the strings' structure as

a stochastic grammar, using a Hidden Markov Model (HMM). In the aforementioned example, the I18N and L10N introduce new, legitimate values in the parameters; thus, the frequency of numbers in $m^{(\text{char})}$ changes and new symbols (e.g., `'-'`, `'[a-zA-Z]'` have to be taken into account. It is straightforward to note that $m^{(\text{struct})}$ is affected in terms of new transitions introduced in the HMM by the new strings. Secondly, request structure changes may affect any type of request model, regardless of the specific characteristics. For instance, if a model for a new parameter is missing, requests that contain that parameter might be flagged as anomalous.

Session changes

Changes in sessions occur whenever resource path sequences are *reordered*, *inserted*, or *removed*. Adding or removing application modules introduces changes in the session models. Also, modifications in the application logic are reflected in the session models as reordering of the resources invoked.

Example. A new version of a web-based community software grants read-only access to anonymous users, allowing them to display contents previously available to subscribed users only. In the old version, legitimate sequences were $\langle$`/site`, `/auth`, `/blog`$\rangle$ or $\langle$`/site`, `/auth`, `/files`$\rangle$, where `/site` indicates the server-side resource that handles the public site, while `/auth` and `/blog` were formerly private resources. Initially, the probability of observing `/auth` before `/blog` or `/files` is close to one (since users need to authenticate before accessing private material). This is no longer true in the new version, however, where `/site` $\rightarrow$ `/files`|`/blog`|`/auth` are all possible.

Changes in sessions impact all models that rely on the sequence of resources that are invoked during the normal operation of an application. For instance, consider the model $m^{(\text{sess})}$, which builds a probabilistic finite state automaton that captures sequences of resource paths. New transitions must be added to take into account the changes mentioned in the above example. These types of models are sensitive to strong changes in the session structure and should be updated accordingly when they occur.

Response changes

Changes in responses occur whenever an application is upgraded to produce different responses. Interface redesigns and feature addition or removal are example causes of changes in the responses. Response changes are frequent, since page updates or redesigns often occur in modern websites.

Example. A new version of a video sharing application introduces Web 2.0 features into the user interface, allowing for the modification of user interface elements without refreshing the entire page. In the old version, relatively few nodes of documents generated by the application contained client-side code. In the new version, however, many nodes of the document contain event handlers to trigger asynchronous requests to the application in response to user events. Thus, if a response model is not updated to reflect the new structure of such documents, a large number of false positives will be generated due to legitimate changes in the characteristics of the web application responses.

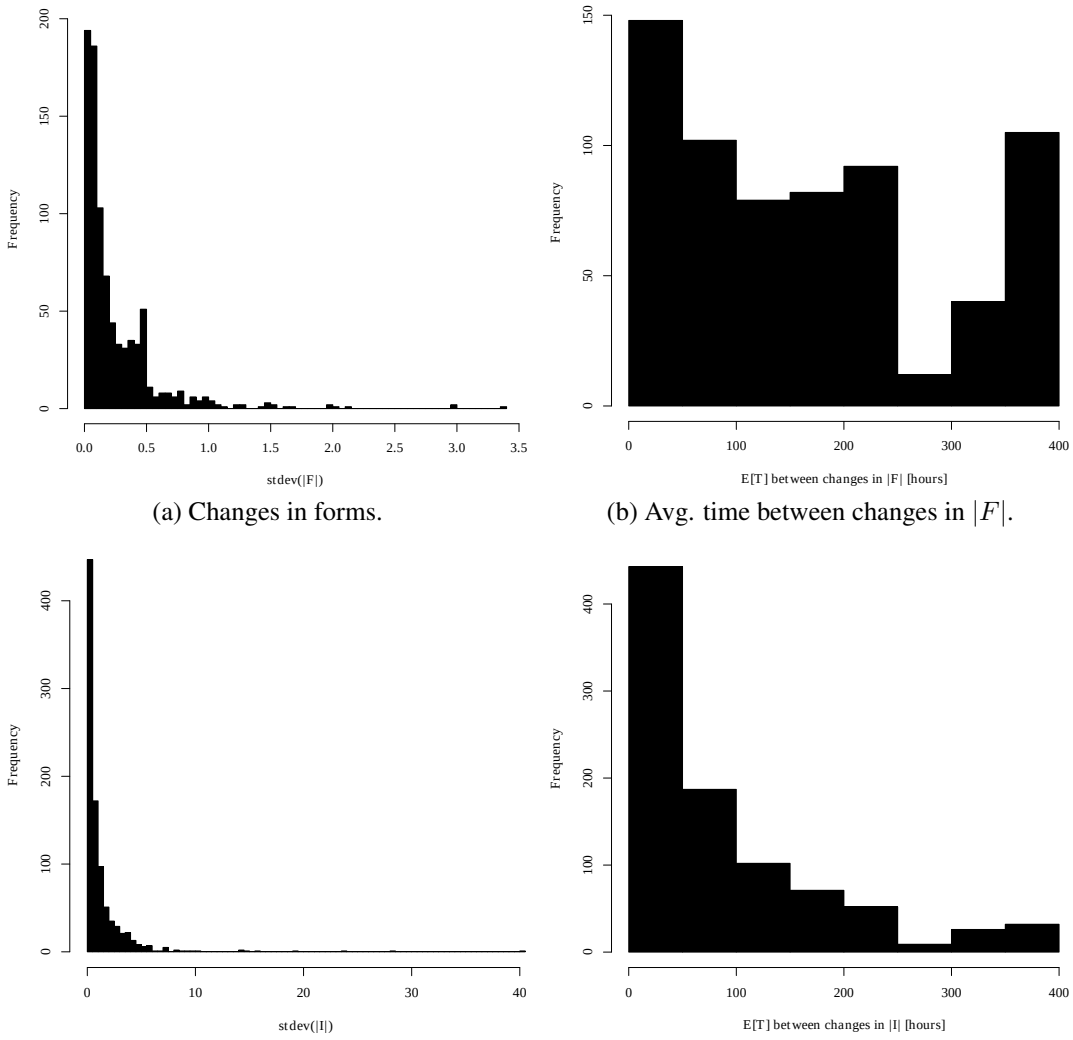
6.1.3 Prevalence of concept drift

To understand whether concept drift is a relevant issue for real-world websites, we performed a number of experiments. For the first experiment, we monitored 2,264 public websites, including the Alexa Top 500 and other sites collected by querying Google with popular terms extracted from the Alexa Top 500. The goal was to identify and quantify the changes in the forms and input fields of popular websites at large. This provides an indication of the frequency with which web applications are updated or altered.

Once every hour, we visited one representative page for each of the 2,264 websites. In total, we collected 3,303,816 pages, comprising more than 1,390 snapshots for each website, between January 29, 2009 and April 13, 2009. 10% of the representative pages were manually selected to have a significant number of forms, input fields, and hyperlinks with parameters. By doing this, we gathered a considerable amount of information regarding the HTTP messages generated by some applications. Examples of these pages are registration pages, data submission pages, or contact form pages. For the remaining websites, we simply used their home pages.

For each website w, each page sample crawled at time t is associated with a tuple $\left\langle |F|_t^{(w)}, |I|_t^{(w)} \right\rangle$, the cardinality of the sets of forms and input fields, respectively. By doing this, we collected samples of the variables

$$|F|^w = \left\{ |F|_{t_1}^w, \ldots, |F|_{t_n}^w \right\}$$



(a) Changes in forms.

(b) Avg. time between changes in $|F|$.

(c) Changes in inputs.

(d) Avg. time between changes in $|I|$

Figure 6.3: Relative frequency of the standard deviation of the number of forms (a) and input fields (c). Also, the distribution of the expected time between changes of forms (b) and input fields (d) are plotted. A non-negligible portion of the websites exhibit changes in the responses.

and

$$|I|^w = \left\{ |I|^w_{t_1}, \ldots, |I|^w_{t_n} \right\},$$

with $0 < n \lesssim 1,390$. Figure 6.3 shows the relative frequency of the variables

$$X_I = \left\{ \text{stdev}(|I|^{(w_1)}), \ldots, \text{stdev}(|I|^{(w_k)}) \right\}$$

and

$$X_F = \left\{ \text{stdev}(|F|^{(w_1)}), \ldots, \text{stdev}(|F|^{(w_k)}) \right\}.$$

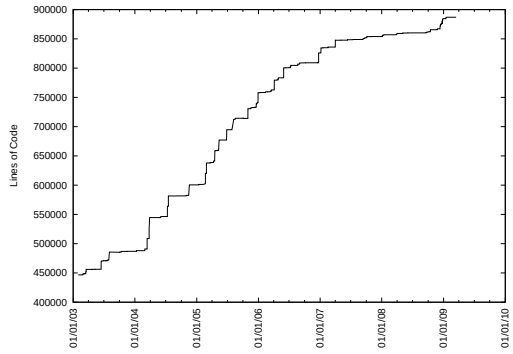
This demonstrates that a significant number of websites exhibit variability in their responses, in terms of elements modified in the pages, as well as requests, in terms of new forms and parameters. In addition, we estimated the expected time between changes of forms and inputs fields, $\mathbb{E}[T_F]$ and $\mathbb{E}[T_I]$, respectively. In terms of forms, 40.72% of the websites changed during the observation period. More precisely, 922 out of 2,264 websites have a finite $\mathbb{E}[T_F]$. Similarly, 29.15% of the websites exhibited changes in the number of input fields – that is, $\mathbb{E}[T_I] < +\infty$ for 660 websites. Figure 6.3 shows the relative frequency of (b) $\mathbb{E}[T_F]$ and (d) $\mathbb{E}[T_I]$. This confirms that a non-negligible portion of the websites exhibit significantly frequent changes in the responses.

For the second experiment, we monitored in depth three large, data-centric web applications over several months: Yahoo! Mail, YouTube, and MySpace. We dumped HTTP responses captured by emulating user interaction using a custom, scriptable web browser implemented with HtmlUnit. Examples of these interactions are as follows:

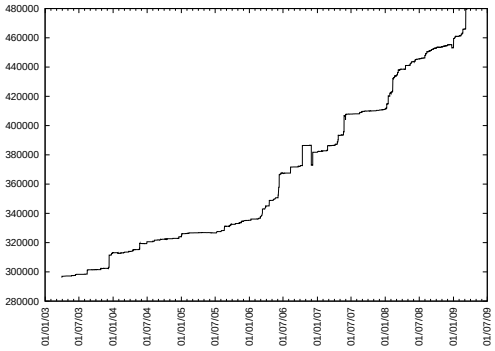
visit the home page, login, browse the inbox, send messages, return to the home page, click links, log out. Manual inspection revealed some major changes in Yahoo! Mail. For instance, the most evident change consisted of a set of new features added to the search engine (e.g., local search, refined address field in maps search). User pages of YouTube were significantly updated with new functionality between 2008 and 2009. Notably, the new version handles several new parameters that allow users to reposition widgets in their personal pages. The analysis on MySpace did not reveal any significant change. The results of these two experiments show that changes in server-side applications are common. More importantly, these modifications often involve the way user data is represented, handled, and manipulated.

For the third experiment, we measured changes in requests and sessions by inspecting the code repositories of three of the largest, most popular open-source web applications: WordPress, Movable Type, and PhpBB. The goal was to understand whether upgrading a web application to a newer release results in significant changes in the features that are used to determine its behavior. In this analysis, we examined changes in the source code that affect the manipulation of HTTP responses, requests, and session data. We used StatSVN[2], a tool for tracking and visualizing the activity of SVN repositories (e.g., the number of lines changed or the most active developers). We modified StatSVN to incorporate a set of heuristics to compute approximate counts of the lines of code that, directly or indirectly, manipulate HTTP session, request or response data. In the case of PHP, examples representative of such lines include, but are not limited to, `_REQUEST|_SESSION|_POST|_GET|session_|-http_|strip_tags|addslashes`. In order to take into account data manipulation

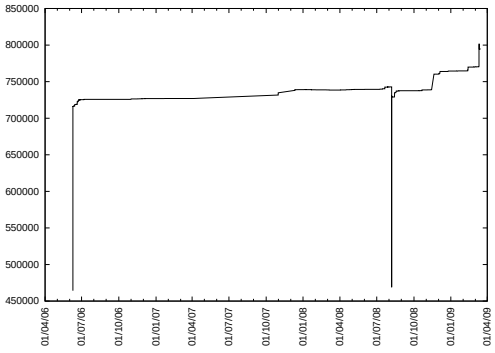
[2]Source available for download at `http://statsvn.org`



(a) PhpBB



(b) WordPress



(c) Movable Type

Figure 6.4: Lines of codes in the repositories of PhpBB, WordPress, and Movable Type, over time. Counts include only the code that manipulates HTTP responses, requests, and sessions.

performed through library functions (e.g., WordPress' custom `Http` class), we also generated application-specific code patterns by manually inspecting and filtering the core libraries. Figure 6.4 shows, over time, the lines of code in the repositories of PhpBB, WordPress, and Movable Type that manipulate HTTP responses, requests and, sessions. These results show the presence of significant modifications in the web application in terms of relevant lines of code added or removed . More importantly, such modifications affect the way HTTP data is manipulated and, thus, impact request, response or session models.

The aforementioned experiments provide evidence that changes in web applications are common, and they affect features used to model the behavior of the applications. Therefore, we conclude that anomaly detectors for web applications must incorporate procedures to prevent false alerts due to concept drift. In particular, a mechanism is needed to discriminate between legitimate and malicious changes and respond accordingly.

6.2 Addressing concept drift

In this section, we first present a technique to distinguish between legitimate changes in web application behavior and evidence of malicious activity. We then discuss how a web application anomaly detection system can effectively handle legitimate concept drift.

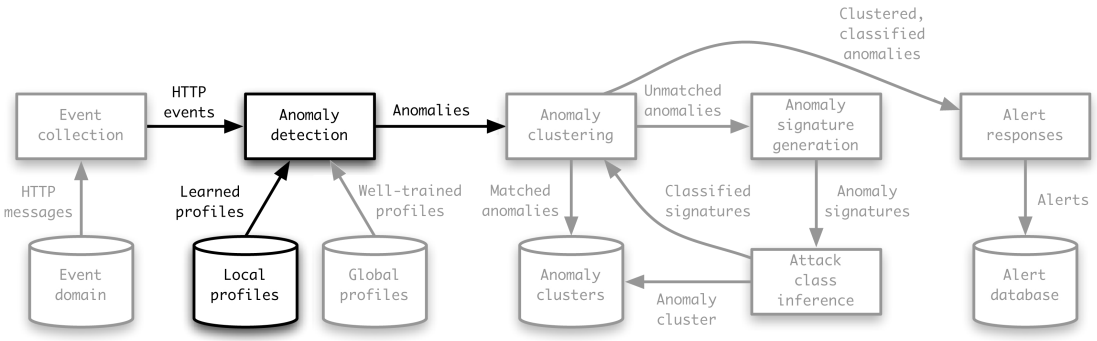


Figure 6.5: Architectural overview of WEBANOMALY, with concept drift components highlighted.

6.2.1 The web application as oracle

The body of HTTP responses contains a set of links L_i and forms F_i that refer to a set of target resources. Each form also includes a set of input fields I_i. In addition, each link $l_{i,j} \in L_i$ and form $f_{i,j} \in F_i$ has an associated set of parameters.

From a resource r_i, the client clicks upon a link $l_{i,j}$ or submits a form $f_{i,j}$. Either of these actions generates a new HTTP request to the web application with a set of parameter key-value pairs, resulting in the return of a new HTTP response to the client, r_{i+1}, the body of which contains a set of links L_{i+1} and forms F_{i+1}. This process continues until the session has ended (i.e., either the user has explicitly logged out, or a timeout has occurred).

In the context of addressing concept drift, our key observation is that, at each step of a web application session, the set of potential target resources is given exactly by the content of the current resource. That is, given r_i, the associated sets of links L_i and forms F_i directly encode the set of possible r_{i+1}. Furthermore, each link $l_{i,j}$ and

form $f_{i,j}$ indicates a precise set of expected parameters and, in some cases, the set of legitimate values for those parameters that can be provided by a client.

Example. Consider a hypothetical banking web application, where the current resource $r_i = /\texttt{account}$ presented to a client is an account overview containing a set of links

$$L_i = \left\{ \begin{array}{r} /\texttt{account/history?aid} = 328849660322, \\ /\texttt{account/history?aid} = 446825759916, \\ /\texttt{account/transfer}, \\ /\texttt{logout} \end{array} \right\},$$

and forms (represented as their target action)

$$F_i = \left\{ \begin{array}{r} /\texttt{feedback}, \\ /\texttt{search} \end{array} \right\}.$$

From L_i and F_i, we can deduce the set of legal candidate resources for the next request r_{i+1}. Any other resource would, by definition, be a deviation from a legal session flow through the web application as specified by the application itself. For instance, it would not be expected behavior for a client to directly access `/account/transfer/submit` (i.e., a resource intended to submit an account funds transfer) from r_i. Furthermore, for the resource `/account/history`, it is clear that the web application expects to receive a single parameter `aid` with an account number as an identifier.

In the case of the form with target `/feedback`, let the associated input elements be:

```
<select name="subject">
<option>General</option>
<option>User interface</option>
<option>Functionality</option>
</select>
<textarea name="message" />
```

It immediately follows that any invocation of the `/feedback` resource from r_i should include the parameters `subject` and `message`. In addition, the legal set of values for the parameter `subject` is given by enumerating the enclosed `<option />` tags. Any deviation from this specification could be considered evidence of malicious behavior.

We conclude that the responses generated by a web application constitute a specification of the intended behavior of clients and the expected inputs to an application's resources. As a consequence, when a change occurs in the interface presented by a web application, this will be reflected in the content of its responses. Therefore, our anomaly detection system performs response modeling to detect and adapt to changes in monitored web applications. The details of this approach are discussed in the following section.

6.2.2 Adaptive response modeling

In order to detect changes in web application interfaces, the response modeling of WEBANOMALY has been augmented with the ability to parse links and forms contained in HTML documents returned to a client. The approach is divided into two phases.

Extraction and parsing

In the first phase, the anomaly detector parses each HTML document contained in a response issued by the web application to a client. For each `<a />` tag encountered, the contents of the `href` attribute is extracted and analyzed. The link is decomposed into tokens representing the protocol (e.g., `http`, `https`, `javascript`, `mailto`), target host, port, path, parameter sequence, and anchor. Paths are subject to additional processing; for instance, relative paths are normalized to obtain a canonical representation. This information is stored as part of an abstract document model for later processing.

A similar process occurs for forms. When a `<form />` tag is encountered, the `action` attribute is extracted and analyzed as in the case of the link `href` attribute. Furthermore, any `<input />`, `<textarea />`, or `<select />` and `<option />` tags enclosed by a particular `<form />` tag are parsed as parameters to the corresponding form invocation. For `<input />` tags, the `type`, `name`, and `value` attributes are extracted. For `<textarea />` tags, the `name` attribute is extracted. Finally, for `<select />` tags, the `name` attribute is extracted, as well as the content of any enclosed `<option />` tags. The target of the form and its parameters are recorded in the abstract document model as in the case for links.

Analysis and modeling

During the second phase, the set of links and forms contained in a response is processed by the anomaly engine. For each link and form, the corresponding target resource is compared to the existing known set of resources. If the resource has not been observed before, a new model is created for that resource. The session model is also updated to account for a potential transition from the resource associated with the parsed document and the target resource by training on the observed session request sequence.

For each of the parameters parsed from links or forms contained in a response, a comparison with the existing set of known parameters is performed. If a parameter has not already been observed, a profile is created and associated with the target resource model.

Any values contained in the response for a given parameter are processed as training samples for the associated models. In cases where the total set of legal parameter values is specified (e.g., `<select />` and `<option />` tags), the parameter profile is updated to reflect this. Otherwise, the profile is trained on subsequent requests to the associated resource.

As a result of this analysis, the anomaly detector is able to adapt to changes in session structure resulting from the introduction of new resources. In addition, the anomaly detector is able to adapt to changes in request structure resulting from the introduction of new parameters and, in a limited sense, to changes in parameter values.

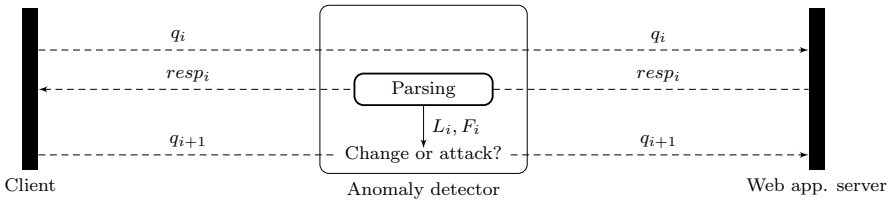


Figure 6.6: An abstract representation of the interaction between the client and the web application server, monitored by a learning-based anomaly detector. After request q_i is processed, the corresponding response $resp_i$ is intercepted and link L_i and forms F_i are parsed to update the request models. This knowledge is exploited as a change detection criterion for the subsequent request q_{i+1}.

6.2.3 Advantages and limitations

Due to the response modeling algorithm described in the previous section, our web application anomaly detector is able to automatically adapt to many common changes observed in web applications as modifications are made to the interface presented to clients. Both changes in session and request structure can be accounted for in an automated fashion. Furthermore, we claim that web application anomaly detectors that do not perform response modeling cannot reliably distinguish between anomalies caused by legitimate changes in web applications and those caused by malicious behavior. Therefore, as will be shown in Section 6.3, any such detector that solely monitors requests is more prone to false positives in the real world.

Clearly, the technique relies upon the assumption that the web application has not been compromised. Since the web application, and in particular the documents it generates, is treated as an oracle for whether a change has occurred, if an attacker were to compromise the application in order to introduce a malicious change, the malicious behavior would be learned as normal by our anomaly detector. Of course, in this case,

the attacker would already have access to the web application. Also, we assume that our anomaly detector observes all requests and responses to and from untrusted clients. Therefore, any attack that would compromise response modeling would be detected and blocked.

Besides the aforementioned assumptions, three limitations are important to note. First, the set of target resources may not always be statically derivable from a given resource. For instance, this can occur when client-side scripts are used to dynamically generate page content, including links and forms. Accounting for dynamic behavior would require the inclusion of script interpretation. This, however, has a high overhead, is complex to perform accurately, and introduces the potential for denial of service attacks against the anomaly detection system. For these reasons, we have not included such a component in the current system, although further research is planned to deal with dynamic behavior. Moreover, as Section 6.3 demonstrates, the proposed technique performs well in practice.

Second, the technique does not fully address changes in the behavior of individual request parameters in its current form. In cases where legitimate parameter values are statically encoded as part of an HTML document, response modeling can directly account for changes in the legal set of parameter values. Unfortunately, in the absence of any other discernible changes in the response, changes in parameter values provided by clients cannot be detected. However, heuristics such as detecting when all clients switch to a new observable behavior in parameter values (i.e., all clients generate anomalies against a set of models in a similar way) could serve as an indication that a change in legitimate parameter behavior has occurred.

Third, the technique cannot handle the case where a resource is the result of a parameterized query and the previous response has not been observed by the anomaly detector. In our experience, however, this does not occur frequently in practice, especially for sensitive resources.

6.3 Evaluation

In this section, we show that our techniques reliably distinguish between legitimate changes and evidence of malicious behavior, and present the resulting improvement in terms of detection accuracy.

The goal of this evaluation is twofold. We first show that concept drift in modeled behavior caused by changes in web applications results in lower detection accuracy. Second, we demonstrate that our technique based on HTTP responses effectively mitigates the effects of concept drift. In both the experiments, the testing data set includes samples of the most common attacks against web applications such as cross-site scripting (XSS), SQL injections, and command execution exploits that are reflected in request parameter values. Representative examples of these attacks include:

- malicious JavaScript inclusion

  ```
  <script src="http://example.com/malware.js"></script>;
  ```

- bypassing login authentication

  ```
  ' OR 'x'='x'--;
  ```

- command injection

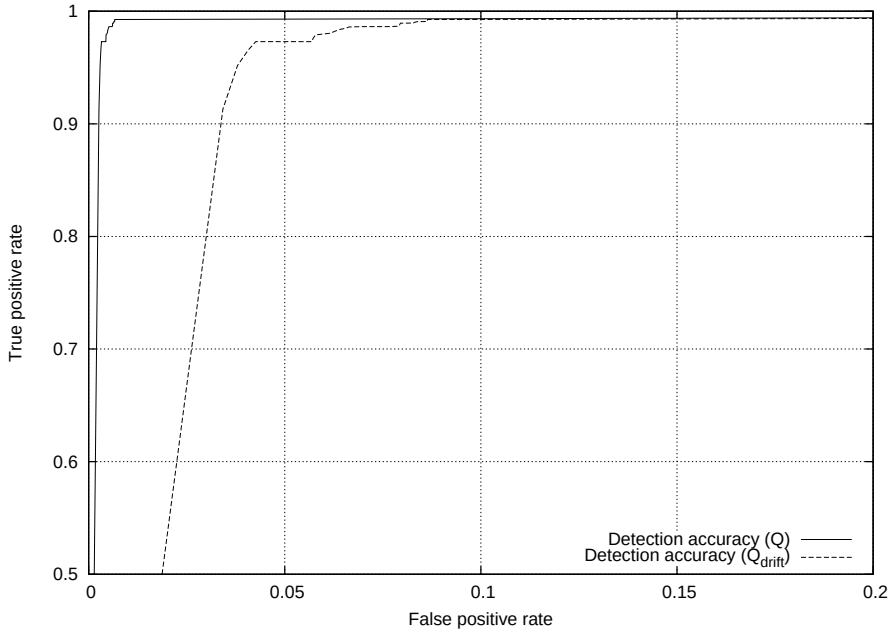
```
; cat /etc/passwd | mail attacker@gmail.com #.
```

In both experiments, WEBANOMALY was evaluated on a data set consisting of HTTP traffic drawn from real-world web applications. This data was obtained from several monitoring points at both commercial and academic sites. For each application, the full contents of each HTTP connection observed over a period of several months were recorded. The resulting flows were filtered using signature-based techniques to remove known attacks, and then partitioned into distinct training and test sets. In total, the data set contains 823 unique web applications, 36,392 unique resource paths, 16,671 unique parameters, and 58,734,624 HTTP requests.

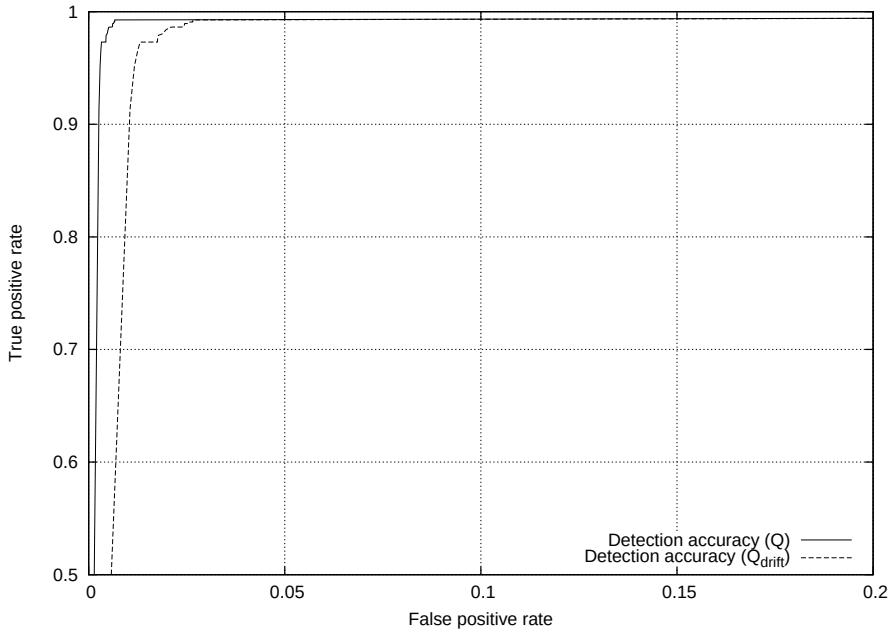
6.3.1 Effects of concept drift

In the first experiment, we demonstrate that concept drift as observed in real-world web applications results in a significant negative impact on false positive rates. First, WEBANOMALY was trained on an unmodified, filtered data set. Then, the detector analyzed a test data set Q to obtain a baseline ROC curve.

After the baseline curve had been obtained, the test data set was processed to introduce new behaviors corresponding to the effects of web application changes, such as upgrades or source code refactoring, obtaining Q_{drift}. In this manner, the set of changes in web application behavior was explicitly known. In particular, new session flows were created by introducing requests for new resources and creating request sequences for both new and known resources that had not previously been observed. Also, new



(a) Response modeling disabled.



(b) Response modeling enabled.

Figure 6.7: True positives plotted against false positives measured on Q and Q_{drift}, with HTTP response modeling enabled in (b).

parameter sets were created by introducing new parameters to existing requests. Finally, the behavior of modeled features of parameter values was changed by mutating observed values in client requests. In all cases, responses generated by the web application were modified to reflect changes in client behavior. For instance, references to new resources were inserted in documents generated by the web application, and both links and forms contained in documents were updated to reflect new parameters.

WEBANOMALY – without the HTTP response modeling technique enabled – was then run over Q_{drift} to determine the effects of concept drift upon detector accuracy. The resulting ROC curves are shown in Figure 6.7a. The consequences of web application change are clearly reflected in the increase in false positive rate for Q_{drift} versus that for Q. Each new session flow and parameter manifests as an alert, since the detector is unable to distinguish between anomalies due to malicious behavior and those due to legitimate changes in the web application.

6.3.2 Change detection

The second experiment quantifies the improvement in the detection accuracy of WEBANOMALY in the presence of web application change. As before, the detector was trained over an unmodified filtered data set, and the resulting profiles were evaluated over both Q and Q_{drift}. In this experiment, however, the HTTP response modeling technique was enabled.

Figure 6.7b presents the results of analyzing HTTP responses on detection accuracy. Since many changes in the behavior of the web application and its clients can be discov-

Change type	Anomalies	False positives	Reduction
Session flow	6,749	0	100.0%
New parameter	6,750	0	100.0%
Modified parameter	5,785	4,821	16.6%
Total	19,284	4,821	75.0%

Table 6.1: Reduction in the false positive rate due to HTTP response modeling for various types of changes.

ered using our response modeling technique, the false positive rate for Q_{drift} is greatly reduced over that shown in Figure 6.7a, and approaches that of Q, where no changes have been introduced. The small observed increase in false positive rate can be attributed to the effects of changes in parameter values. This occurs because a change has been introduced into a parameter value submitted by a client to the web application, and no indication of this change was detected on the preceding document returned to the client.

Table 6.1 displays the individual contributions to the reduction of the false positive rate due to the response modeling technique. Specifically, the total number of anomalies caused by each type of change, the number of anomalies erroneously reported as alerts, and the corresponding reduction in the false positive rate is shown. The results displayed were generated from a run using the optimal operating point indicated by the knee of the ROC curve in Figure 6.7b. For changes in session flows and parameters sets, the detector was able to identify an anomaly as being caused by a change in web application behavior in all cases. This resulted in a large net decrease in the false positive rate of the detector with response modeling enabled. The modification of parameters is more problematic, though; as discussed in Section 6.2.3, it is not always

apparent that a change has occurred when that change is limited to the type of behavior a parameter's value exhibits. We note, however, that drastic changes in the behavior of a given parameter are uncommon in practice. Rather, it is more often the case that a new parameter is introduced, and, therefore, this result should be considered a worst-case evaluation.

From the overall improvement in false positive rates, we conclude that HTTP response modeling is an effective technique for distinguishing between anomalies due to legitimate changes in web applications and those caused by malicious behavior. Furthermore, any anomaly detector that does not do so is prone to generating a large number of false positives when changes do occur in the modeled application. Finally, as it has been shown in Section 6.1, web applications exhibit significant long-term change in practice, and, therefore, concept drift is a critical aspect of web application anomaly detection that must be addressed.

6.4 Conclusions

In this chapter, we have identified the natural dynamicity of web applications as an issue that must be addressed by modern learning-based web application anomaly detectors. Otherwise, increases in the false positive rate of an anomaly detector will result whenever the monitored web application is changed. We refer to this phenomenon as *web application concept drift*.

We presented an empirical study of the degree of change exhibited by real-world web

applications over an extended period of time, and found that substantial changes indeed occur. We then demonstrated the use of novel HTTP response modeling techniques in WEBANOMALY to discriminate between legitimate changes and anomalous behaviors in web applications. More precisely, responses are analyzed to detect legitimate changes in web application behavior in terms of the interfaces presented to web clients. This information is then leveraged to update the corresponding request and session models. Finally, we evaluated the effectiveness of our approach over an extensive real-world data set of web application traffic. The results show that WEBANOMALY enhanced with response modeling can effectively distinguish between anomalies and legitimate changes, avoiding false alerts in the presence of concept drift.

As future work, we plan to investigate the potential benefits of modeling the behavior of JavaScript code, which is becoming increasingly prevalent in modern web applications. Also, additional, richer, and media-dependent response models must be studied to account for Rich Internet Applications, such as Adobe Flash and Microsoft Silverlight applications.

Chapter 7

Static Enforcement of Web Application Integrity

The main focus of this dissertation to this point has been on machine learning-based techniques for performing anomaly detection of web-based attacks. In general, much research has focused on lightweight, black-box methods for detecting and preventing the exploitation of security vulnerabilities in web applications. This approach has many benefits when applied to existing applications, where the cost of re-architecting a complex application would be prohibitive, or the source code for an application is not available. Anomaly detection approaches in particular are attractive due to their black-box approach, since they typically need no *a priori* knowledge of the structure or implementation of a web application in order to provide effective detection. The viability of this approach for protecting web applications is evidenced by the proliferation of web application firewalls (WAFs) that incorporate some form of anomaly-based detection

techniques [11, 31, 15, 66, 106].

As mentioned in Chapter 1, there exist a number of other approaches to securing software besides detection techniques. In particular, another significant focus of research has been on applying various static and dynamic analyses to the source code of web applications in order to identify and mitigate security vulnerabilities prior to deployment [46, 78, 54, 7, 14, 135]. These approaches have the advantage that developers can continue to create web applications using traditional languages and development frameworks, and periodically apply a vulnerability analysis tool to provide a level of assurance that no security-relevant flaws are present. Analyzing web applications is a complex task, however, as is the interpretation of the results of such security tools. Additionally, several approaches require developers to specify security policies to be enforced in a specialized language.

A more recent line of research has focused on providing client-side protection by enforcing security policies within the web browser [101, 50, 27]. These approaches show promise in detecting and preventing client-side attacks against newer web applications that aggregate content from multiple third parties, but the specification of policies to enforce is generally left to the developer.

In this chapter, we propose a different, preventative approach to web application security, one that constitutes a radical departure from the anomaly detection system described heretofore. We observe that cross-site scripting (XSS) and SQL injection vulnerabilities can be viewed as a failure on the part of the web application to enforce a separation of the *structure* and the *content* of documents and database queries, and that this is a result of treating documents and queries as untyped sequences of bytes. There-

fore, instead of protecting or analyzing existing web applications, we describe a framework that strongly types both documents and database queries. The framework is then responsible for automatically enforcing a separation between structure and content, as opposed to the *ad hoc* sanitization checks that developers currently must implement. Consequently, the integrity of documents and queries generated by web applications developed using our framework are automatically protected, and thus, *by construction*, such web applications are not vulnerable to server-side cross-site scripting and SQL injection attacks.

To illustrate the problem at hand, consider that HTML or XHTML documents to be presented to a client are typically constructed by concatenating strings. Without additional type information, a web application framework has no means of determining that the following operations could lead to the introduction of a cross-site scripting vulnerability:

```
String result = "<div>" + userInput + "</div>";
```

The key intuition behind the framework presented in this chapter is that because both documents and database queries are strongly typed in our framework, the framework can distinguish between the structure (`<div>` and `</div>`) and the content (`userInput`) of these critical objects, and enforce their integrity automatically. To accomplish this, we leverage the advanced type system of Haskell, since it offers a natural means of expressing the typing rules we wish to impose. In principle, however, a similar framework could be implemented in any language with a strong type system that allows for some form of multiple inheritance (e.g., Java or C#).

In this chapter, we present the design of a strongly typed web application framework that can be used to create new web applications that are secure by construction. This framework automatically prevents the introduction of server-side cross-site scripting and SQL injection vulnerabilities by strongly typing both web documents and database queries. We then evaluate the design of the framework, demonstrating the coverage and correctness of its sanitization functions, and conclude that web applications developed under this framework are free from certain classes of vulnerabilities.[1]

7.1 Framework design

At a high level, the web application framework is composed of several familiar components. A web server component processes HTTP requests from web clients and forwards these requests in an intermediate form to the application server based on one of several configuration parameters (e.g., URL path prefix). These requests are directed to one of the web applications hosted by the application server. The web application examines any parameters to the request, performs some processing during which queries to a back-end database may be executed, and generates a document. Note that in the following, the terms "document" or "web document" shall generically refer to any text formatted according to the HTML or XHTML standards. This document is then returned down the component stack to the web server, which sends the document as part of an HTTP response to the web client that originated the request. A graphical depiction of this architecture is given in Figure 7.1.

[1] An earlier version of this work appeared in [105].

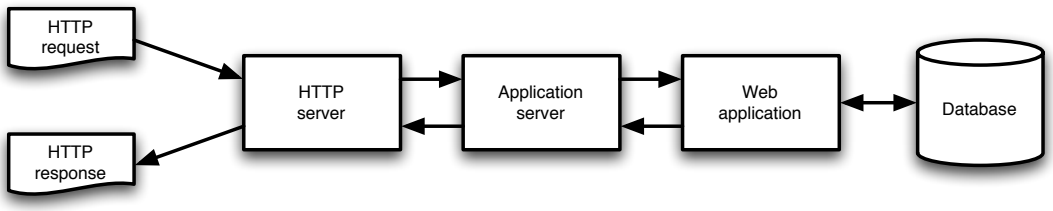


Figure 7.1: Architectural overview of the web application framework.

Web applications developed for our framework are structured as a set of functions with access to a combination of configuration data and application state. More precisely, web applications execute inside the `App` monad. Monads are a category theoretic construction that have found wide application in Haskell to sequence computations or isolate code that can produce side effects.[2] For the purposes of our framework, we use the `App` monad to thread implicit state through the functions comprising a web application. This monad is also used to provide a controlled interface to potentially dangerous functions, effectively sandboxing applications under the framework. In particular, the `App` monad itself is structured as a stack of monad transformers that provide a functional interface to a read-only configuration type `AppConfig`, a read-write application state type `AppState`, and filtered access to the `IO` monad. The definitions for `AppConfig` and `AppState` are given in Figures 7.2 and 7.3.

The `AppConfig` type holds static information relating to the configuration of the application, including the port on which to listen for HTTP requests and the root directory of static files to serve from the filesystem. Of particular interest, however, are the `RouteMap` and `StmtMap` types. The `RouteMap` type describes how URL paths are mapped to values of type `DocumentGen`, which are simply functions that generate

[2]For further information on monads, please refer to [87, 128].

```
data AppConfig = AppConfig {
     appCfgPort :: Int,
     appCfgPrefix :: String,
     appCfgRoutes :: RouteMap,
     appCfgFileRoot :: FilePath,
     appCfgDBConn :: Connection,
     appCfgDBStmts :: StmtMap
}
```

Figure 7.2: Definition for the `AppConfig` type.

documents within the `App` monad. In addition, the `RouteMap` type contains a default `DocumentGen` type that specifies an error page. Given an incoming HTTP request destined for a particular web application, the application server uses that application's `RouteMap` type to determine the proper function to call in order to generate the document to be returned to the client.[3] Finally, the `StmtMap` type associates unique database query identifiers to prepared statements that can be executed by a document generator.

```
data AppState = AppState {
     appStClient :: Maybe SockAddr,
     appStUrl :: Maybe Url
}
```

Figure 7.3: Definition for the `AppState` type.

The `AppState` type contains mutable state that is specific to each request for a document. In particular, one field records information indicating the source of the request. Additionally, another field records the URL that was requested, including any parameters that were specified by the client. More complex state types that hold additional information (e.g., cached database queries or documents) are possible, however.

[3]This construction is similar to the "routes" packages present in popular web development frameworks such as Rails [40] and Pylons [9].

7.2 Document structure

In this section, we introduce the means by which documents are specified under the framework. Then, we discuss how these specifications allow the framework to automatically contain the potentially harmful effects of dynamic data.

7.2.1 Document specification

Once an appropriate route from the `RouteMap` structure has been selected by the application server, the associated document generator function is executed within the context of the `App` monad (i.e., with access to the configuration and current state of the application). The document generator function processes the request from the application server and returns a variable of type `Document`. The definition of the `Document` type and its constituent types are shown in Figure 7.4.

As a result, documents in our framework are not represented as an unstructured stream of bytes. Rather, the structure of the `Document` type closely mirrors that of parsed HTML or XHTML documents. The `DocumentType` field indicates the document's type, such as "HTML 4.01 Transitional" or "XHTML 1.1". The `DocumentHead` type contains information such as the title and client-side code to execute. Finally, the `DocumentBody` type contains a single field that represents the root of a tree of nodes that represent the body of the document.

Each node in this tree is an instantiation of the `Node` type. Each `Node` instantiation maps to a distinct (X)HTML element, and records the set of possible properties of

```
data Document Document {
     docType :: DocumentType,
     docHead :: DocumentHead,
     docBody :: DocumentBody
}

data DocumentType = DOC_TYPE_HTML_4_01_STRICT
                  | DOC_TYPE_HTML_4_01_TRANS
                  | ...
                  | DOC_TYPE_XHTML_1_1

data DocumentHead = DocumentHead {
     docTitle :: String,
     docLinks :: [Node],
     docScripts :: [Node],
     docBaseUrl :: Maybe Url,
     docBaseTarget :: Maybe Target,
     docProfile :: [Url]
}

data DocumentBody = DocumentBody {
     docBodyNode :: Node
}
```

Figure 7.4: Definition for the `Document` type.

that element. For instance, the `TextNode` data constructor creates a `Node` that holds a text string to be displayed as part of a document. The `AnchorNode` data constructor, on the other hand, creates a `Node` that holds information such as the `href` attribute, `rel` attribute, and a list of child nodes corresponding to the text or other elements that comprise the "body" of the link. A partial definition of the `Node` type is presented in Figure 7.5.

With this construction, the entire document produced by a web application in our framework is strongly typed. Instead of generating a document as a byte stream, document

```
data Node = TextNode {
     nodeText :: String
}         | AnchorNode {
     anchorAttrs :: NodeAttrs,
     anchorHref :: Maybe Url,
     anchorRel :: Maybe Relationship,
     anchorRev :: Maybe Relationship,
     anchorTarget :: Maybe Target,
     anchorType :: Maybe MimeType,
     anchorCharset :: Maybe CharSet,
     anchorLang :: Maybe Language,
     anchorName :: Maybe AttrValue,
     anchorShape :: Maybe Shape,
     anchorCoords :: Maybe Coordinates,
     anchorNodes :: [Node]
}         | DivNode {
     divAttrs :: NodeAttrs,
     divNodes :: [Node]
} ...
```

Figure 7.5: Sample `Node` definitions.

structure is explicitly encoded as a tree of nodes. Furthermore, each element and element attribute has an associated type that constrains, to one degree or another, the range of possible values that can be represented. For instance, the `MimeType`, `CharSet`, and `Language` types are examples of enumerations that strictly limit the set of possible values the attribute can take to legal values. Standard (X)HTML element attributes (e.g., `id`, `class`, `style`) are represented with the `NodeAttr` type. Optional attributes are represented using either the `Maybe` type,[4] or as an empty list if multiple elements are allowed.

Note that it is possible for a `Document` to represent an (X)HTML document that is not

[4] `Maybe` allows for the absence of a value, as Haskell does not possess nullable types. For example, the type `Maybe a` can be either `Just "..."` or `Nothing`.

necessarily consistent with the respective W3C grammars that specify the set of well-formed documents. One example is that any `Node` instantiation may appear as the child of any other `Node` that can hold children, which violates the official grammars in several instances. Strict conformance with the W3C standards is not, however, our goal.[5]

Instead, the typing scheme presented here allows our framework to specify a separation between the *structure* and the *content* of the documents a web application generates. More precisely, the dynamic data that enters a web application as part of an HTTP request (e.g., as a GET or POST parameter) can indirectly influence the structure of a document. For instance, a search request to a web application may result in a variable number of table rows in the generated document depending on the number of results returned from a database query. Due to our framework, however, client-supplied data cannot *directly* modify the structure of the document in such a way that a code injection can occur.

7.2.2 Enforcing document integrity

Once a `Document` has been constructed by the web application in response to a client request, it is returned to the application server. The application server is responsible for converting this data structure into a format the client can understand – that is, it must *render* the document into a stream of bytes representing an (X)HTML document. Consequently, the set of types that can comprise a `Document` are instances of the `Render`

[5] Indeed, standards-conforming documents have been shown to be difficult to represent in a functional language [26].

```
class Render a where
      render :: a -> String

instance Render AttrValue where
         render = quoteAttr

quoteAttr :: AttrValue -> String
quoteAttr a = foldl' step [] (attrValue a)

step acc c | c == '<' = acc ++ "&lt;"
           | c == '>' = acc ++ "&gt;"
           | c == '&' = acc ++ "&amp;"
           | c == '"' = acc ++ "&quot;"
           | otherwise = acc ++ [c]
```

Figure 7.6: `Render` typeclass definition and simplified instance example. Here, `quoteAttr` performs a left fold over attribute values using `foldl'`, which applies the `step` function to each character of the string and accumulates the result. The definition of `step` specifies a number of *guards*, where `| c == '<'` is a condition that must be satisfied for the statement `acc ++ "<"` to execute. This statement simply appends the string `"<"` to `acc`, the accumulator, in order to build a new, sanitized string. If no guard condition is satisfied, the character is appended without conversion.

typeclass, shown in Figure 7.6.[6]

The `Render` type class specifies that any instance of the class must implement the `render` function. From the type signature, the semantics of the function are clear: `render` converts an instance type into a string representation suitable for presentation to a client. For our purposes, the `Render` type class is also responsible for enforcing the integrity of a document's structure.

As an example, Figure 7.6 presents a simplified `render` definition for the `AttrValue` type that is used to indicate element attribute strings that may assume (almost) arbitrary

[6]Haskell typeclasses are roughly similar to Java interfaces, in that they specify a function interface that all instances (in Java, implementors) must provide.

values. In order to preserve the integrity of the document, an attribute value must not contain certain characters that would allow an attacker to inject malicious code into the document. Consider, for instance, the following element:

```
<input type="hidden" name="h1" value="..."/>
```

Now, suppose an attacker submitted the following string as part of a request such that it was reflected to another client as the value of the hidden input field:

```
"/>
<script src="http://example.com/malware.js">
</script>
<span id="
```

The result would be the following:

```
<input type="hidden" name="h1" value=""/>
<script src="http://example.com/malware.js">
</script>
<span id=""/>
```

To prevent such an injection from occurring, the `render` function for the `AttrValue` class applies a sanitization function on the string wrapped by `AttrValue`. Any occurrence of an unsafe character is replaced by an equivalent HTML entity encoding that

can safely appear as part of an attribute value.[7] Similar `render` functions are defined for the set of types that can comprise a `Document`.

Therefore, to prepare a `Document` as part of an HTTP response to a client, the application server applies the `render` function to the document, which recursively converts the data structure into an (X)HTML document. As part of this process, the content of the document is sanitized by type-specific `render` functions, ensuring that client-supplied input to the web application cannot modify the document structure in such a way as to result in a client-side code injection.

7.3 SQL query structure

Similar to the case of documents, SQL queries are given structure in our framework through the application of strong typing rules that control how the structure of the query can be combined with dynamic data. In this section, we examine the structure of SQL queries and discuss two mechanisms by which SQL query integrity is enforced under the framework.

7.3.1 Query specification

SQL queries, as shown in Figure 7.7, are composed of clauses, predicates, and expressions. For instance, a clause might be `SELECT *` or `UPDATE users`. An example of

[7] In the real implementation, the sanitization function is somewhat more complex, as there are multiple encodings by which an unsafe character can be injected. The example function given here is simplified for the purposes of presentation.

```
INSERT INTO users(login, passwd) VALUES(?, ?)
SELECT * FROM users WHERE login='admin' AND passwd='test'
UPDATE users SET passwd='$passwd' WHERE login='$login'
```

Figure 7.7: Examples of SQL queries.

a predicate is `login='admin'`, where `'admin'` is an expression. Clauses, predicates, and expressions are themselves composed of static tokens, such as keywords (`SELECT`) and operators (`=`), and dynamic tokens, such as table identifiers (`users`) or data values (`'admin'`).

Typically, the structure of a SQL query is fixed.[8] Specifically, a query will have a static keyword denoting the operation to perform, will reference a static set of tables and fields, and specify a fixed set of predicates. Generally, the only components of a query that change from one execution to the next are data values, and, even then, their number and placement remain fixed.

SQL injection attacks rely upon the ability of the attacker to modify the structure of a query in order to perform a malicious action. When SQL queries are constructed using string operations without sufficient sanitization applied to user input, such attacks become trivial. For instance, consider the `UPDATE` query shown in Figure 7.7. If an attacker were to supply the value "`quux' OR login='admin`" for the `$login` variable, the following query would result:

```
UPDATE users SET passwd='foo' WHERE login='quux' OR login='admin'
```

[8]This is not always the case, but the case of dynamic query structure will be considered later in this section.

```
SELECT * FROM users WHERE login=? AND passwd=?
UPDATE users SET passwd=? WHERE login=?
```

Figure 7.8: Examples of prepared statements, where "?" characters serve as placeholders for data substitution.

Because the attacker was able to inject single quotes, which serve as delimiters for data values, the structure of the query was changed, resulting in a privilege escalation attack.

7.3.2 Integrity enforcement with static query structure

In contrast to the case of document integrity enforcement, a well-known solution exists for specifying SQL query structure: prepared statements. Prepared statements are a form of database query consisting of a parameterized query template containing placeholders where dynamic data should be substituted. An example is shown in Figure 7.8, where the placeholders are signified by the "?" character.

A prepared statement is typically parsed and constructed prior to execution, and stored until needed. When an actual query is to be issued, variables that may contain client-supplied data are *bound* to the statement. Since the query has already been parsed and the placeholders specified, the structure of the query cannot be modified by the traditional means of providing malicious input designed to be interpreted as part of the query. In the case of the injection attack described previously, the result would be the following (note that the injected single quotes have been escaped):

```
UPDATE users SET passwd='foo' WHERE login='quux'' OR login=''admin'
```

From our perspective, the query has been typed as a composition of static and dynamic elements; it is exactly this distinction between structure and content that we wish to enforce. Haskell's database library (HDBC) supports the use of prepared statements, as do most other database libraries. Therefore, the framework exports functions that allow a web application to associate prepared statements with a unique identifier in the `AppConfig` type. During request processing, a document generator can then retrieve a prepared statement using the identifier, bind values to it, and execute queries that are not vulnerable to injection attacks.

One detail remains, however. The HDBC library also provides functions that allow traditional *ad hoc* queries that are assembled as concatenations of strings to be executed. Without any other modification to the framework, a web application developer would be free to directly call these functions and bypass the protections afforded by the framework. Therefore, an additional component is required to encapsulate the HDBC interface and prevent execution of these unsafe functions. This component takes the form of a monad transformer `AppIO`, which simply wraps the `IO` monad and exposes only those functions that are considered safe to execute. The structure of this stack is shown in Figure 7.9. In this environment, within which all web applications using the framework operate, unsafe database execution functions are inaccessible, since they will fail to type-check. Thus, assuming the correctness of the HDBC prepared statement interface, web applications developed using the framework are not vulnerable to SQL injection.

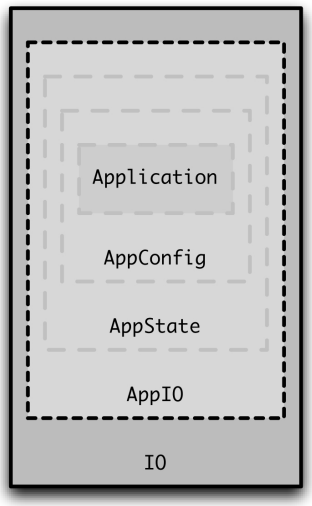


Figure 7.9: Graphical representation of the `App` monad stack within which framework applications execute. The `AppIO` monad encapsulates applications, preventing them from calling unsafe functions within the `IO` monad.

7.3.3 Integrity enforcement with dynamic query structure

Though most SQL queries possess a fixed structure, there does exist a small class of SQL queries that exhibit dynamic structure. For instance, many SQL database implementations provide a set membership operator, where queries of the form

```
SELECT * FROM users WHERE login IN ('admin', 'developer', 'tester')
```

can be expressed. In this case, the size of the set of data values can often change at runtime. Another example is the case where the structure of queries is determined by the user, for instance through a custom search form where many different combinations of predicates can be dynamically expressed. Unfortunately, since these queries cannot be represented using prepared statements, they cannot be protected using the monadic encapsulation technique described previously.

Therefore, a second database interface is exposed by the framework to the application developers. Instead of relying upon prepared statements, this interface allows developers to dynamically construct queries as a tree of algebraic data types as in the case of web documents. Figure 7.10 shows an example of the type representing a `SELECT` query.

To populate instances of these types, the interface provides a set of combinators, or higher-order functions, that can be chained together. These combinators, which assume names similar to SQL keywords, implement an embedded domain-specific language (DSL) that allows application developers to naturally specify dynamic queries within the framework. For instance, a query could be constructed using the following sequence of function invocations:

```
qSelect [qField "*"] >>=
qFrom [qTable "users"] >>=
qWhere (((qField "login") == (qData "admin")) &&
((qField "passwd") == (qData "test")))
```

Similar to the case of the `Document` type, queries constructed in this manner are transformed into raw SQL statements solely by the framework.[9] Therefore, the types that represent queries also implement the `Render` typeclass. Consequently, sanitization functions must be applied to each of the fields comprising the query types, such that the intended structure of the query cannot be modified. This can be accomplished by enforcing the conditions that no data value may contain an unescaped single quote,

[9]Note that, as in the case for web documents, we do not attempt to enforce the generation of correct SQL, but rather focus on preventing attacks by preserving query structures specified by the developer.

```
data Select = Select {
     sFields :: [Expr],
     sTables :: [Expr],
     sCons :: Maybe Expr,
     sGrpFields :: [Expr],
     sGrpCons :: Maybe Expr,
     sOrdFields :: [Expr],
     sLimit :: Maybe Int,
     sOffset :: Maybe Int,
     sDistinct :: Bool
}

data Expr = EXPR_TABLE Table
          | EXPR_FIELD Field
          | EXPR_DATA String
          | EXPR_NOT Expr
          | EXPR_OR Expr Expr
          | EXPR_AND Expr Expr
          | ...

data Table = Table {
     tName :: String,
     tAlias :: Maybe String
}

data Field = Field {
     fName :: String,
     fAlias :: Maybe String
}
```

Figure 7.10: Definition for the `Select` type.

and that all remaining query components may not contain spaces, single quotes, or characters signifying the beginning of a comment. Assuming that these sanitization functions are correct, this construction renders applications developed under the framework invulnerable to SQL injection attacks while allowing for more powerful query specifications.

7.4 Evaluation

To demonstrate that web applications developed using our framework are secure by construction from server-side XSS and SQL injection vulnerabilities, we conducted an evaluation of the system. First, we demonstrate that all dynamic content contained in a `Document` must be sanitized by an application of the `render` function, and that a similar condition holds for dynamically-generated SQL queries. Then, we provide evidence that the sanitization functions themselves are correct – that is, they successfully strip or encode unsafe characters. We also verify that the prepared statement library prevents injections, as expected. Finally, to demonstrate the viability of the framework, an experiment to evaluate the performance of a web application developed using the framework is conducted.

7.4.1 Sanitization function coverage

The goal of the first experiment was to justify the claim that all dynamic content contained in a `Document` or query type must be sanitized prior to presentation to the client

that originated the request. To accomplish this, a static control flow analysis of the framework was performed. Figure 7.11 presents a control flow graph of the application server in a simplified form, where function calls are sequenced from left to right. Of particular interest is the `renderDoc` function, which retrieves the appropriate document generator given a URL path, executes it in the call to `route`, sanitizes it by applying `render`, and creates an HTTP response by calling `make200`. The sanitized document is then returned to `procRequest`, which writes it to the client. Therefore, the entire process of converting the document to a byte stream for presentation to the client is solely due to the recursive `render` application. Similarly, because the only interface exposed to applications to execute SQL queries are `execStmt` and `execPrepStmt` from within the `App` monad, queries issued by applications under the framework must be sanitized either by the framework or the HDBC prepared statement functions.

Figure 7.12 displays a subset of the full control flow graph depicting an instance of the `render` function for the `AnchorNode Node` instantiation. For clarity of presentation, multiple calls to `render` and `maybeRenderAttr` have been collapsed into single nodes. Recall from Figure 7.5 that the definition of `AnchorNode` does not contain any bare strings; instead, each field of the type is either itself a composite type, or an enumeration for which a custom `render` function is defined. Since no other string conversion function is applied in this subgraph, we conclude that all data contained in an `AnchorNode` variable must be filtered through a sanitization function.

The analysis of this single case generalizes to the set of all types that can comprise a `Document` or query type. In total, 163 distinct sanitization function definitions were checked to sanitize the contexts shown in Table 7.1. For each function, our analysis

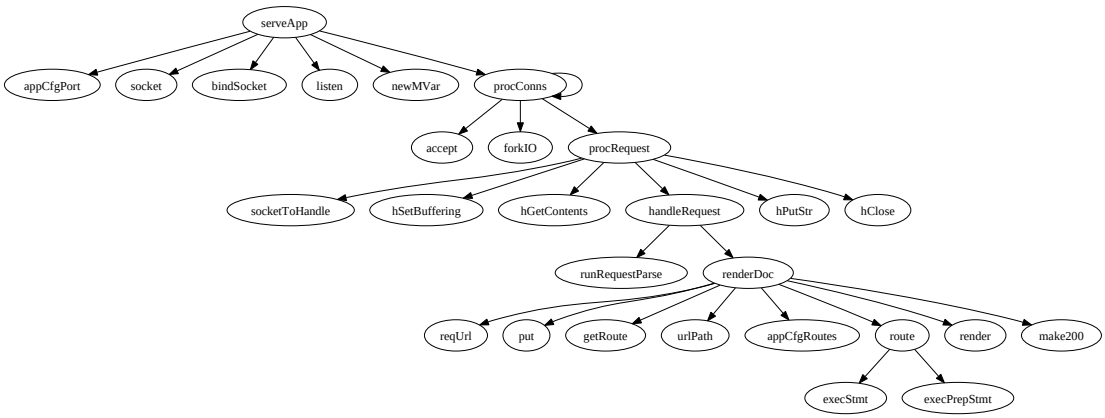


Figure 7.11: Simplified control flow graph for application server.

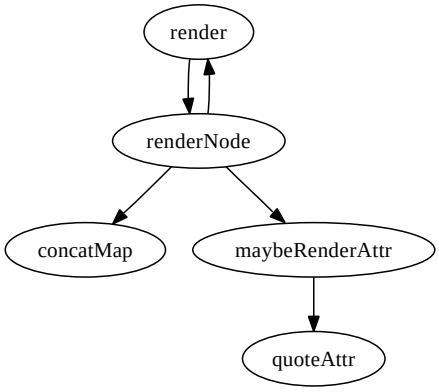


Figure 7.12: Example control flow graph for `Render Node` instance.

found that no irreducible type was concatenated to the document byte stream without first being sanitized.

7.4.2 Sanitization function correctness

The goal of this experiment is to determine whether the sanitization functions employed by the framework are correct (i.e., whether all known sequences of dangerous characters are stripped or encoded). To establish this, we applied a dynamic test-driven

Context	Semantics
Document nodes	Conversion to static string
Document node attributes	Encoding of HTML entities
Document text	Encoding of HTML entities
URL components	Encoding of HTML entities, percent encoding
SQL static values	Removal of spaces, comments, quotes
SQL data values	Escaping of quotes

Table 7.1: Example contexts for which specific sanitization functions are applied, and the semantics of those sanitization functions under various encodings.

approach using the QuickCheck property testing library [16]. QuickCheck allows a developer to use an embedded language to specify invariants that should hold for a given set of functions. The library then automatically generates a set of random test cases, and checks that the invariants hold for each test. In our case, we selected invariants based upon known examples of XSS [103] and SQL injection attacks [32]. In addition, we introduced modifications of the invariants that account for different popular document encodings, since these encodings directly affect how browser parsers interpret the sequences of bytes that comprise a document.

Since the coverage of the sanitization functions has been established by the control flow analysis, we focused our invariant testing on the low-level functions responsible for processing string data. In particular, we specified invariants for 7 functions that are responsible for sanitizing (X)HTML content, element attributes, and various URL components.[10] An example invariant specification is shown in Figure 7.13.

For each of the sanitization functions, we first tested the correctness of the invariants by checking that they were violated over a set of 100 strings corresponding to real-world

[10]The 163 functions noted above eventually apply one of these 7 context-specific sanitization functions for web documents.

```
propAttrValueSafe :: AttrValue -> Bool
propAttrValueSafe input =
    (not $ elem '<' output) &&
    (not $ elem '>' output) &&
    (not $ elem '&' $ stripEntities output) &&
    (not $ elem '"' output) where
    output = render input
```

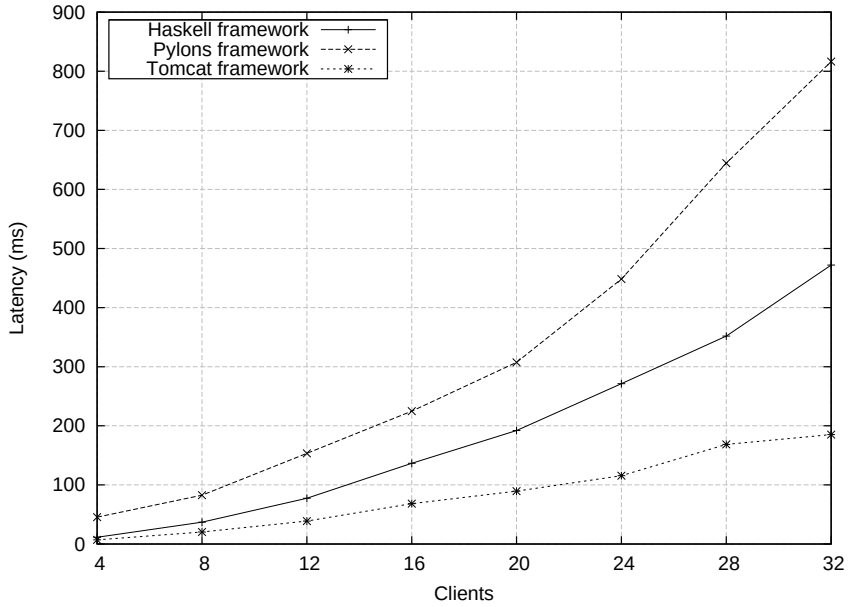
Figure 7.13: Simplified sanitization function invariant specification. Here, `propAttrValueSafe` is a conjunction of predicates, where `not $ elem c output` specifies that the character `c` should not be an element of the output of `render` in this context. Since "`&`" is used to indicate the beginning of an HTML entity (e.g., `&`), the `stripEntities` function ensures that ampersands may only appear in this form.

cross-site scripting, command injection, and other code injection attacks. Then, for each sanitization function, we generated 1,000,000 test cases of random strings using the QuickCheck library. In all cases, the invariants were satisfied.

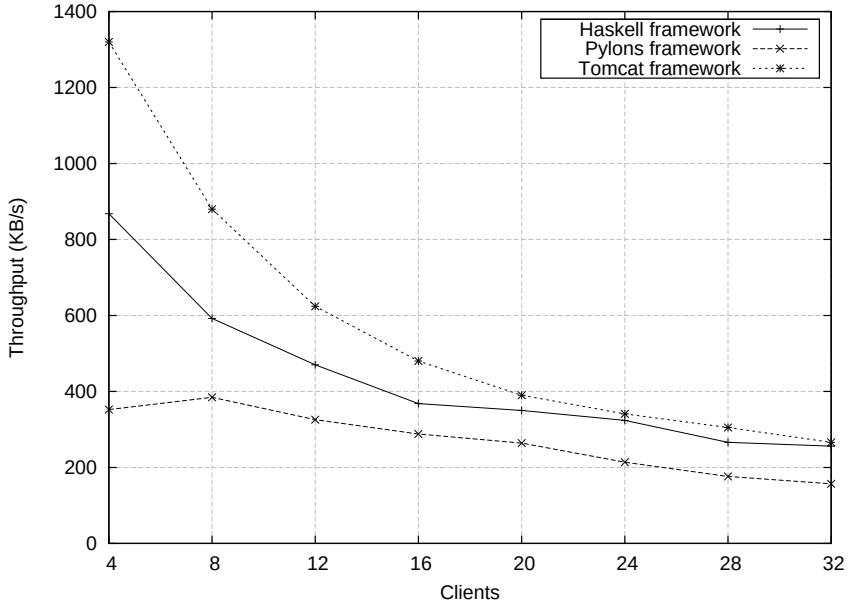
In addition to performing invariant testing on the set of document sanitization functions, we also applied a similar testing process to the sanitization of query types described in Section 7.3.3. Finally, we applied manual invariant testing on the HDBC prepared statement interface. In all cases, the invariants on the integrity of the queries and the database itself held.

7.4.3 Framework performance

In this experiment, we compared the performance of a web application developed using our framework to similar applications implemented using other frameworks. In particular, we developed a small e-commerce site with a product display page, cart display



(a) Latency as a function of concurrent clients.



(b) Throughput as a function of concurrent clients.

Figure 7.14: Latency and throughput performance for the Haskell, Pylons, and Tomcat-based web applications.

page, and checkout page under our framework, using the Pylons framework 0.9.7 [9], and as a Java servlet using Tomcat 6.0.18.[11] Each application was backed by a SQLite database containing product information. The application servers were hosted on a server running Ubuntu Server 8.10 with dual Intel Core 2 Duo CPUs, 2 GB of RAM, and 1000BaseT Ethernet network interfaces. The `httperf` [43] web server benchmarking tool was deployed on a similar server to generate load for each application.

Figure 7.14 presents averaged latency and throughput plots for 8 benchmarking runs for each framework tested. In each run, the number of concurrent clients issuing requests was varied, and the average response latency in milliseconds and the aggregate throughput in kilobytes was recorded. In this experiment, our framework performed competitively compared to Pylons and Tomcat, performing somewhat better than Pylons in both latency and throughput scaling, and vice versa for Tomcat. In particular, the latency plot shows that our framework scales significantly better with the number of clients than the Pylons framework. Unfortunately, our framework exhibited approximately a factor of two increase in latency compared to the Tomcat application. Cost-center profiling revealed that this is mainly due to the overhead of list-based `String` operations in Haskell,[12] though this could be ameliorated by rewriting the framework to prefer the lower-overhead `ByteString` type. Therefore, it is not unreasonable to assume that web applications developed using our framework would exhibit acceptable performance behavior in the real world.

[11] Pylons is a Python-based framework that is similar in design to Ruby on Rails, and is used to implement a variety of well-known web applications (e.g., Reddit (`http://reddit.com/`)).

[12] Strings are represented as lists of characters in Haskell – that is, `type String = [Char]`.

7.4.4 Discussion

The security properties enforced by this framework are effective at guaranteeing that applications are not vulnerable to server-side XSS and SQL injection. There are limitations to this protection that need to be highlighted, however, and we discuss these here.

First, web applications can, in some cases, be vulnerable to *client-side* XSS injections, or DOM-based XSS, where the web application can potentially not receive any portion of such an attack [60]. This can occur when a client-side script dynamically updates the DOM after the document has been rendered by the browser with data controlled by an attacker. In general, XSS attacks stemming from the misbehavior of client-side code within the browser are not addressed by the framework in its current form.

Recently, a new type of XSS attack against the content-sniffing algorithms employed by web browsers has been demonstrated [10]. In this attack, malicious non-HTML files that nevertheless contain HTML fragments and client-side code are uploaded to a vulnerable web application. When such a file is downloaded by a victim, the content-sniffing algorithm employed by the victim's browser can potentially interpret the file as HTML, executing the client-side code contained therein, resulting in a XSS attack. Consequently, our framework implements the set of file upload filters recommended by the authors of [10] to prevent content-sniffing XSS. Since, however, the documents are supplied by users and not generated by the framework itself, the framework cannot guarantee that it is immune to such attacks.

Finally, CSS stylesheets and JSON documents can also serve as vectors for XSS at-

tacks. In principle, these documents could be specified within the framework using the same techniques applied to (X)HTML documents, along with context-specific sanitization functions. In the case of CSS stylesheets that are uploaded to a web application by users, additional sanitization functions could be applied to strip client-side code fragments. However, the framework in its current form does not address these vectors.

7.5 Conclusions

In this chapter, we have presented a framework for developing web applications that, by construction, are invulnerable to server-side cross-site scripting and SQL injection attacks. The framework accomplishes this by strongly typing both documents and database queries that are generated by a web application, thereby automatically enforcing a separation between structure and content that preserves the integrity of these objects.

We conducted an evaluation of the framework, and demonstrated that all dynamic data that is contained in a document generated by a web application must be subjected to sanitization. Similarly, we showed that all SQL queries are executed in a safe manner. We also demonstrated the correctness of the sanitization functions themselves. Finally, we gave performance numbers for representative web applications developed using this framework that compare favorably to those developed in other popular environments.

In future work, we plan to investigate how the framework can be modified to allow developers to specify "safe" transformations of document structure that occur in a con-

trolled manner. Specifically, this would allow for dynamic changes to document structures to occur in response to user data that are not vulnerable to XSS injections. Also, we plan to investigate static techniques for verifying the correctness of the sanitization functions in terms of their agreement with invariants extracted from web browser document parsers and database query parsers, for instance using a combination of static and dynamic analyses [8, 10]. Finally, future work will consider how language-based techniques for ensuring document integrity could be applied on the client.

Chapter 8

Conclusions

The number and severity of security incidents on the World Wide Web are both increasing, and methods to mitigate the pervasive vulnerabilities present in the Web is an active area of research. Anomaly detection techniques are a promising avenue for both providing advanced monitoring capabilities as well as offering the ability to block web attacks. Unfortunately, existing anomaly detection approaches suffer from several significant drawbacks. This dissertation has outlined the design of WEBANOMALY, a next-generation web application anomaly detection system that is intended to address these deficiencies.

To accomplish this, a number of novel contributions to the area of web application anomaly detection have been presented. In particular, new models for characterizing normal sequences of web client queries and the structure of documents generated by web applications have been discussed. The use of *anomaly signatures* to reduce false

positive rates and provide attack classification were introduced. An approach to addressing a scarcity of training data in the context of web applications by leveraging global similarities between application features was presented. Finally, we propose a technique to adapt to *concept drift*, or changes in the behavior of web application features over time, by treating the web application as an oracle of change. Each of these techniques were introduced as components of WEBANOMALY, and shown to dramatically increase the detection accuracy of the system as well as reduce its false positive rate.

Though many obstacles to the effective use of anomaly detection techniques for web-based attacks have been surmounted, several challenges remain. In particular, the increasing prevalence of client-side code, such as JavaScript and Rich Internet Frameworks, increases the attack surface of web applications and introduces a new set of potential vulnerabilities. These must be addressed by anomaly detection systems in order to provide acceptable coverage of monitored web applications.

Anomaly detection techniques are a lightweight and effective solution for protecting existing applications. New web applications, however, offer an opportunity to secure those applications against known attacks from the design phase. To that end, this dissertation has also presented a framework for developing web applications that are secure against cross-site scripting (XSS) and SQL injection attacks by construction. This is accomplished by strongly typing both documents and database queries issued by the web application, allowing for the framework to automatically enforce a separation between *structure* and *content*. The resulting framework, implemented in Haskell, was evaluated to verify the correctness and coverage of its santization functions. In addi-

tion, an evaluation was conducted of its performance relative to existing web application development frameworks, and the Haskell-based framework was found to compare favorably.

Though XSS and SQL injection are by far the most serious web application vulnerabilities today, numerous other types of attacks abound that are not addressed by this framework. Future work will focus on automatic mitigation of these attacks. Also, while adoption of Haskell and other functional languages continues to grow, it is foreseeable that many applications will continue to be developed using imperative languages. Therefore, future work will examine how the techniques that have been proposed in this dissertation may be applied in an imperative context.

Bibliography

[1] M. Almgren and U. Lindqvist. Application-Integrated Data Collection for Security Monitoring. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2001)*, LNCS, pages 22–36, Davis, CA, USA, October 2001. Springer.

[2] P. Ammann, S. Jajodia, and P. Liu. Recovery From Malicious Transactions. *IEEE Transactions on Knowledge and Data Engineering*, 15(5):1167–1185, September 2002.

[3] K. Ashcraft and D. Engler. Using Programmer-Written Compiler Extensions to Catch Security Holes. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2002)*, page 143, Oakland, CA, USA, May 2002. IEEE Computer Society.

[4] S. Axelsson. The Base-Rate Fallacy and its Implications for the Difficulty of Intrusion Detection. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 1999)*, pages 1–7, New York, NY, USA, 1999. ACM.

[5] T. Ball and S. K. Rajamani. Automatically Validating Temporal Safety Properties of Interfaces. In *Proceedings of the International Workshop on Model Checking of Software (SPIN 2001)*, LNCS, pages 103–122. Springer, 2001.

[6] D. Balzarotti, M. Cova, V. V. Felmetsger, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Saner: Composing Static and Dynamic Analysis to Validate Sanitization in Web Applications. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, Oakland, CA, USA, May 2008.

[7] D. Balzarotti, M. Cova, V. V. Felmetsger, and G. Vigna. Multi-module Vulnerability Analysis of Web-based Applications. In *Proceedings of the 2007 ACM Conference on Computer and Communications Security (CCS 2007)*, Alexandria, VA, USA, October 2007.

[8] D. Balzarotti, W. Robertson, C. Kruegel, and G. Vigna. Improving Signature Testing Through Dynamic Data Flow Analysis. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2007)*, Miami Beach, FL, USA, December 2007.

[9] B. Bangert and J. Gardner. PylonsHQ. `http://pylonshq.com/`, February 2009.

[10] A. Barth, J. Caballero, and D. Song. Secure Content Sniffing for Web Browsers, or How to Stop Papers from Reviewing Themselves. In *Proceedings of the IEEE Symposium on Security and Privacy*, Oakland, CA, USA, May 2009. IEEE Computer Society.

[11] Breach Security, Inc. Breach WebDefend. `http://www.breach.com/products/webdefend.html`, January 2009.

[12] C. Cadar, P. Twohey, V. Ganesh, and D. Engler. EXE: A System for Automatically Generating Inputs of Death Using Symbolic Execution. In *Proceedings of the Conference on Computer and Communications Security (CCS 2002)*, Alexandria, VA, USA, October 2006. ACM.

[13] H. Chen and D. Wagner. MOPS: An infrastructure for examining security properties of software. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2002)*, pages 235–244, Washington, DC, USA, October 2002. ACM.

[14] S. Chong, K. Vikram, and A. C. Myers. SIF: Enforcing Confidentiality and Integrity in Web Applications. In *Proceedings of the 2007 USENIX Security Symposium*, Boston, MA, USA, 2007. USENIX Association.

[15] Citrix Systems, Inc. Citrix Application Firewall. `http://www.citrix.com/English/PS2/products/product.asp?contentID=25636`, January 2009.

[16] K. Claessen and J. Hughes. Testing Monadic Code with QuickCheck. *ACM SIGPLAN Notices*, 37(12):47–59, 2002.

[17] W. Cohen, P. Ravikumar, and S. Fienberg. A comparison of string distance metrics for name-matching tasks. In *Proceedings of the IJCAI-2003 Workshop on Information Integration on the Web (IIWeb-03)*, 2003.

[18] Coverity, Inc. Coverity static source code analysis. http://www.coverity.com/, June 2008.

[19] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the USENIX Security Conference (USENIX Security 1998)*, pages 63–78, San Antonio, TX, USA, January 1998. USENIX Association.

[20] G. F. Cretu, A. Stavrou, M. E. Locasto, S. J. Stolfo, and A. D. Keromytis. Casting out Demons: Sanitizing Training Data for Anomaly Sensors. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (S&P 2008)*, pages 81–95, Oakland, CA, USA, May 2008. IEEE Computer Society.

[21] M. de Kunder. The Size of the World Wide Web. http://www.worldwidewebsize.com/, June 2009.

[22] D. E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.

[23] Django Software Foundation. Django Web Application Framework. `http://www.djangoproject.com/`, June 2009.

[24] N. Een and N. Sörensson. An Extensible SAT-solver. http://www.minisat.se/, 2003.

[25] M. Egele, C. Kruegel, E. Kirda, H. Yin, and D. Song. Dynamic Spyware Analysis. In *Proceedings of the USENIX Annual Technical Conference (USENIX 2007)*, pages 233–246, Santa Clara, CA, USA, June 2007. USENIX Association.

[26] M. Elsman and K. F. Larsen. Typing XHTML Web Applications in ML. In *Proceedings of the* 6^{th} *International Symposium on Practical Aspects of Declarative Languages*, pages 224–238. Springer-Verlag, 2004.

[27] U. Erlingsson, B. Livshits, and Y. Xie. End-to-end Web Application Security. In *Proceedings of the* 11^{th} *USENIX Workshop on Hot Topics in Operating Systems*, San Diego, CA, USA, 2007. USENIX Association.

[28] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Transactions on Software Engineering*, 27(2):99–123, February 2001.

[29] H. J. Escalante and O. Fuentes. Kernel Methods for Anomaly Detection and Noise Elimination. In *Proceedings of the International Conference on Computing (CORE 2006)*, pages 69–80, Mexico City, Mexico, 2006.

[30] H. Etoh. SSP: GCC extension for protecting applications from stack-smashing attacks. http://www.trl.ibm.com/projects/security/ssp/, June 2008.

[31] F5 Networks, Inc. BIG-IP Application Security Manager. `http://www.f5.com/products/big-ip/product-modules/application-security-manager.html`, January 2009.

[32] Ferruh Mavituna. SQL Injection Cheat Sheet. `http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/`, June 2009.

[33] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. RFC 2616: Hypertext Transfer Protocol – HTTP/1.1. http://tools.ietf.org/rfc/rfc2616.txt, June 1999.

[34] M. Finifter, A. Mettler, N. Sastry, and D. Wagner. Verifiable Functional Purity in Java. In *Proceedings of the* 15^{th} *ACM Conference on Computer and Communications Security*, pages 161–174, Alexandria, VA, USA, October 2008. ACM.

[35] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A Sense of Self for Unix Processes. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 1996)*, pages 120–128, Oakland, CA, USA, May 1996. IEEE Computer Society.

[36] Fortify Software, Inc. Fortify Software vulnerability analysis. http://www.fortify.com/, June 2008.

[37] V. Frias-Martinez, S. J. Stolfo, and A. D. Keromytis. Behavior-Profile Clustering for False Alert Reduction in Anomaly Detection Sensors. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2008)*, Anaheim, CA, USA, December 2008.

[38] A. K. Ghosh, J. Wanken, and F. Charron. Detecting Anomalous and Unknown Intrusions Against Programs. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 1998)*, pages 259–267, Phoenix, AX, USA, December 1998.

[39] Google, Inc. ctemplate. `http://code.google.com/p/google-ctemplate/`, June 2009.

[40] D. H. Hansson. Ruby on Rails. `http://rubyonrails.org/`, February 2009.

[41] P. Helman and G. Liepins. Statistical Foundations of Audit Trail Analysis for the Detection of Computer Misuse. *IEEE Transactions on Software Engineering*, 19(9):886–901, 1993.

[42] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software Verification with Blast. In *Proceedings of the International Workshop on Model Checking of Software (SPIN 2003)*, LNCS, pages 235–239. Springer, 2003.

[43] Hewlett Packard Development Company, L.P. httperf. `http://www.hpl.hp.com/research/linux/httperf/`, February 2009.

[44] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[45] G. J. Holzmann. The Model Checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, 1997.

[46] Y.-W. Huang, F. Yu, C. Hang, C.-H. Tsai, D. Lee, and S.-Y. Kuo. Securing Web Application Code by Static Analysis and Runtime Protection. In *Proceedings of the* 13^{th} *International Conference on the World Wide Web*, pages 40–52, New York, NY, USA, 2004. ACM.

[47] K. Ilgun. USTAT: A Real-Time Intrusion Detection System for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 1993)*, pages 16–28, Oakland, CA, USA, May 1993. IEEE Computer Society.

[48] K. L. Ingham, A. Somayaji, J. Burge, and S. Forrest. Learning DFA Representations of HTTP for Protecting Web Applications. *Computer Networks*, 51(5):1239–1255, April 2007.

[49] H. S. Javitz and A. Valdes. The SRI IDES Statistical Anomaly Detector. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 1991)*, pages 316–326, Oakland, CA, USA, May 1991. IEEE Computer Society.

[50] T. Jim, N. Swamy, and M. Hicks. Defeating Script Injection Attacks with Browser-Enforced Emdedded Policies. In *Proceedings of the* 16^{th} *International Conference on the World Wide Web*, Banff, Alberta, Canada, May 2007. ACM.

[51] M. Johns and C. Beyerlein. SMask: Preventing Injection Attacks in Web Applications by Approximating Automatic Data/Code Separation. In *Proceedings of ACM Symposium on Applied Computing*, Seoul, Korea, March 2007. ACM.

[52] N. D. Jones and N. Andersen. Flow analysis of lazy higher-order functional programs. *Theoretical Computer Science*, 375(1–3):120–136, 2007.

[53] N. Jovanovic, E. Kirda, and C. Kruegel. Preventing Cross Site Request Forgery Attacks. In *Proceedings of the IEEE International Conference on Security and Privacy for Emerging Areas in Communication Networks (SecureCOMM 2006)*, Baltimore, MD, USA, August 2006. IEEE Computer Society.

[54] N. Jovanovic, C. Kruegel, and E. Kirda. Pixy: A Static Analysis Tool for Detecting Web Application Vulnerabilities. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2006)*, pages 258–263, Oakland, CA, USA, May 2006. IEEE Computer Society.

[55] N. Jovanovic, C. Kruegel, and E. Kirda. Precise Alias Analysis for Static Detection of Web Application Vulnerabilities. In *Proceedings of the 2006 Workshop*

on Programming Languages and Analysis for Security, pages 27–36, Ottawa, Ontario, Canada, 2006. ACM.

[56] B. Juang and L. Rabiner. A probabilistic distance measure for hidden Markov models. *AT&T Bell Laboratories Technical Journal*, 64(2):391–408, 1985.

[57] M. Kendall. A new measure of rank correlation. *Biometrika*, 30:81–93, 1938.

[58] S.-i. Kim and N. Nwanze. Noise-Resistant Payload Anomaly Detection for Network Intrusion Detection Systems. In *Proceedings of the Performance, Computing and Communications Conference (IPCCC 2008)*, pages 517–523, Austin, TX, USA, December 2008. IEEE Computer Society.

[59] E. Kirda, C. Kruegel, G. Vigna, and N. Jovanovic. Noxes: A Client-side Solution for Mitigating Cross-Site Scripting Attacks. In *Proceedings of the 2006 ACM Symposium on Applied Computing (SAC 2006)*, Dijon, France, April 2006. ACM.

[60] A. Klein. DOM Based Cross Site Scripting or XSS of the Third Kind. `http://www.webappsec.org/projects/articles/071105.shtml`, July 2005.

[61] C. Ko. Logic Induction of Valid Behavior Specifications for Intrusion Detection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2000)*, pages 142–153, Oakland, CA, USA, May 2000. IEEE Computer Society.

[62] C. Ko, M. Ruschitzka, and K. Levitt. Execution Monitoring of Security-Critical Programs in Distributed Systems: A Specification-based Approach. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 1997)*, pages 175–187, Oakland, CA, USA, May 1997. IEEE Computer Society.

[63] O. Kolesnikov, D. Dagon, and W. Lee. Advanced Polymorphic Worms: Evading IDS by Blending in with Normal Traffic. In *Proceedings of the USENIX Security Symposium (USENIX Security 2006)*, Vancouver, B.C., CA, August 2006. USENIX Association.

[64] J. Kolter and M. Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *The Journal of Machine Learning Research*, 8:2755–2790, 2007.

[65] C. Kruegel, D. Mutz, W. Robertson, and F. Valeur. Bayesian Event Classification for Intrusion Detection. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, Las Vegas, NV, USA, December 2003.

[66] C. Kruegel, W. Robertson, and G. Vigna. A Multi-model Approach to the Detection of Web-based Attacks. *Journal of Computer Networks*, 48(5):717–738, July 2005.

[67] C. Kruegel, T. Toth, and E. Kirda. Service-Specific Anomaly Detection for Network Intrusion Detection. In *Proceedings of the Symposium on Applied Computing (SAC 2002)*, Spain, March 2002.

[68] C. Kruegel and G. Vigna. Anomaly Detection of Web-based Attacks. In *Proceedings of the ACM Conference on Computer and Communications Security (ACM CCS)*, Washington, DC, USA, October 2003. ACM.

[69] S. Kumar and E. H. Spafford. A Pattern Matching Model for Misuse Intrusion Detection. In *Proceedings of the National Computer Security Conference*, pages 11–21, 1994.

[70] M. S. Lam, M. Martin, B. Livshits, and J. Whaley. Securing Web Applications with Static and Dynamic Information Flow Tracking. In *Proceedings of the 2008 ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation*, pages 3–12, San Francisco, CA, USA, 2008. ACM.

[71] T. Lane and C. E. Brodley. Temporal Sequence Learning and Data Reduction for Anomaly Detection. *ACM Transactions on Information and System Security*, 2(3):295–331, 1999.

[72] W. Lee and S. J. Stolfo. A Framework for Constructing Features and Models for Intrusion Detection Systems. *ACM Transactions on Information and System Security*, 3(4):227–261, 2000.

[73] W. Lee and D. Xiang. Information-Theoretic Measures for Anomaly Detection. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2001)*, pages 130–143, Oakland, CA, USA, May 2001. IEEE Computer Society.

[74] P. Li and S. Zdancewic. Encoding Information Flow in Haskell. In *Proceedings of the* 19^{th} *IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2006.

[75] U. Lindqvist and P. A. Porras. Detecting Computer and Network Misuse Through the Production-Based Expert System Toolset (P-BEST). In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 1999)*, pages 146–161, Oakland, CA, USA, May 1999. IEEE Computer Society.

[76] R. P. Lippmann, D. J. Fried, I. Graf, J. W. Haines, K. R. Kendall, D. McClung, D. Weber, S. E. Webster, D. Wyschogrod, R. K. Cunningham, and M. A. Ziss-

man. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition (DISCEX 2000)*, volume 2, pages 12–26, Hilton Head, SC, USA, January 2000.

[77] B. Livshits and U. Erlingsson. Using Web Application Construction Frameworks to Protect Against Code Injection Attacks. In *Proceedings of the 2007 Workshop on Programming Languages and Analysis for Security*, pages 95–104, San Diego, CA, USA, 2007. ACM.

[78] B. Livshits and M. Lam. Finding Security Errors in Java Programs with Static Analysis. In *Proceedings of the* 14^{th} *USENIX Security Symposium (USENIX Security 2005)*, pages 271–286. USENIX Association, August 2005.

[79] T. F. Lunt. Detecting Intruders in Computer Systems. In *Proceedings of the Annual Symposium and Technical Displays on Physical and Electronic Security*, August 1990.

[80] R. B. Lyngsø, C. N. S. Pedersen, and H. Nielsen. Metrics and similarity measures for hidden markov models. In *Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology*, pages 178–186. AAAI Press, 1999.

[81] A. Madhavapeddy, A. Ho, T. Deegan, D. Scott, and R. Sohan. Melange: Creating a "Functional Internet". In *Proceedings of the* 2^{nd} *ACM European Conference on Computer Systems*, pages 101–114, Lisbon, Portugal, April 2007. ACM.

[82] F. Maggi, M. Matteucci, and S. Zanero. Detecting intrusions through system call sequence and argument analysis. *IEEE Transactions on Dependable and Secure Computing*, 99(1), 5555.

[83] F. Maggi, W. Robertson, C. Kruegel, and G. Vigna. Protecting a Moving Target: Addressing Web Application Concept Drift. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID)*, Saint-Malo, FR, September 2009. Springer.

[84] J. McHugh. Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed By Lincoln Laboratory. *ACM Transactions on Information and System Security*, 3(4):262–294, November 2000.

[85] Microsoft, Inc. LINQ. `http://msdn.microsoft.com/en-us/netframework/aa904594.aspx`, June 2009.

[86] Miniwatts Marketing Group. World Internet Usage Statistics. `http://www.internetworldstats.com/stats.htm`, June 2009.

[87] E. Moggi. Notions of Computation and Monads. *Information and Computation*, 93(1):55–92, 1991.

[88] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel. Anomaly system call detection. *ACM Transactions on Information and System Security*, 9(1):61–93, February 2006.

[89] Y. Nadji, P. Saxena, and D. Song. Document Structure Integrity: A Robust Basis for Cross-site Scripting Defense. In *Proceedings of the Network and Distributed System Security Symposium*, February 2009.

[90] Netscape Communications, Inc. Client Side State – HTTP Cookies. http://wp.netscape.com/newsref/std/cookie_spec.html, 1999.

[91] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shifley, and D. Evans. Automatically Hardening Web Applications Using Precise Tainting. In *Proceedings of the 2005 International Information Security Conference*, pages 372–382, 2005.

[92] Ofer Shezaf and Jeremiah Grossman. Web Hacking Incidents Database. http://www.xiom.com/whid-about, June 2009.

[93] Open Security Foundation. DLDOS: Data Loss Database – Open Source. http://datalossdb.org/, June 2009.

[94] Open Web Application Security Project (OWASP). OWASP Top 10 2007. `http://www.owasp.org/index.php/Top_10_2007`, February 2009.

[95] OpenBSD developers. The OpenBSD Project. http://www.openbsd.org/, June 2008.

[96] PaX developers. The PaX Project. http://pax.grsecurity.net/, June 2008.

[97] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. *Journal of Computer Networks*, 31(23-24):2435–2463, December 1999.

[98] PCI Security Standards Council, LLC. PCI Data Security Standard (PCI DSS). https://www.pcisecuritystandards.org/, June 2009.

[99] T. H. Ptacek and T. N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., January 1998.

[100] N. J. Puketza, K. Zhang, M. Chung, B. Mukherjee, and R. A. Olsson. A Methodology for Testing Intrusion Detection Systems. *IEEE Transactions on Software Engineering*, 22(10):719–729, September 1996.

[101] C. Reis, J. Dunagan, H. J. Wang, and O. Dubrovsky. BrowserShield: Vulnerability-Driven Filtering of Dynamic HTML. *ACM Transactions on the Web*, 1(3):11, 2007.

[102] I. Ristic. mod_security: Open Source Web Application Firewall. http://www.modsecurity.org/, June 2008.

[103] Robert Hansen (RSnake). XSS (Cross Site Scripting) Cheat Sheet. `http://hackers.org/xss.html`, June 2009.

[104] W. Robertson, C. Kruegel, D. Mutz, and F. Valeur. Run-time Detection of Heap-based Overflows. In *Proceedings of the USENIX Large Installation Systems Administration Conference (LISA 2003)*, San Diego, CA, USA, October 2003. USENIX Association.

[105] W. Robertson and G. Vigna. Static Enforcement of Web Application Integrity Through Strong Typing. In *Proceedings of the USENIX Security Symposium*, Montreal, CA, August 2009. USENIX Association.

[106] W. Robertson, G. Vigna, C. Kruegel, and R. A. Kemmerer. Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web

Attacks. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2006)*, San Diego, CA, USA, February 2006.

[107] M. Roesch. Lightweight Intrusion Detection for Networks. In *Proceedings of the USENIX Large Installation Systems Administration Conference (LISA 1999)*. USENIX Association, November 1999.

[108] S. Romanosky, R. Telang, and A. Acquisti. Do Data Breach Disclosure Laws Reduce Identity Theft? In *Proceedings of the Workshop on the Economics of Information Security (WEIS 2008)*, Hanover, NH, USA, June 2008.

[109] J. Schlimmer and R. Granger. Beyond incremental processing: Tracking concept drift. In *Proceedings of the Fifth National Conference on Artificial Intelligence*, volume 1, pages 502–507, 1986.

[110] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni. A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2001)*, pages 144–155, Oakland, CA, USA, May 2001. IEEE Computer Society.

[111] J. Seward and N. Nethercote. Using Valgrind to detect undefined value errors with bit-precision. In *Proceedings o the USENIX Annual Technical Conference (USENIX 2006)*, page 2, Boston, MA, USA, May 2005. USENIX Association.

[112] S. Sidiroglou, O. Laadan, A. D. Keromytis, and J. Nieh. Using Rescue Points to Navigate Software Recovery. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2007)*, pages 273–280, Oakland, CA, USA, May 2007. IEEE Computer Society.

[113] Y. Song, A. D. Keromytis, and S. J. Stolfo. Spectrogram: A Mixture-of-Markov-Chains Model for Anomaly Detection in Web Traffic. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2009)*, San Diego, CA, USA, February 2009.

[114] A. Stolcke and S. Omohundro. Hidden Markov Model Induction by Bayesian Model Merging. *Advances in Neural Information Processing Systems*, pages 11–11, 1993.

[115] A. Stolcke and S. Omohundro. Inducing Probabilistic Grammars by Bayesian Model Merging. *Lecture Notes in Computer Science*, pages 106–106, 1994.

[116] A. Stolcke and S. M. Omohundro. Best-first Model Merging for Hidden Markov Model Induction. Technical Report TR-94-003, ICSI, Berkeley, CA, USA, 1994.

[117] B. Stone-Gross, M. Cova, L. Cavallaro, B. Gilbert, M. Szydlowski, R. Kemmerer, C. Kruegel, and G. Vigna. Your Botnet is My Botnet: Analysis of a Botnet Takeover. Technical report, UC Santa Barbara, April 2009.

[118] Symantec, Inc. Symantec Report on the Underground Economy – July 07 – June 08. `http://eval.symantec.com/mktginfo/enterprise/white_papers/b-whitepaper_underground_economy_report_11-2008-14525717.en-us.pdf`, November 2008.

[119] Tenable Network Security, Inc. Nessus network vulnerability scanner. http://www.nessus.org/, June 2008.

[120] The MITRE Corporation. Common Exposures and Vulnerabilities Database. http://cve.mitre.org/, May 2008.

[121] T. Toth and C. Kruegel. Accurate Buffer Overflow Detection via Abstract Payload Execution. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2002)*, Switzerland, October 2002. Springer.

[122] A. Valdes and K. Skinner. Adaptive, Model-based Monitoring for Cyber Attack Detection. In H. Debar, L. Me, and F. Wu, editors, *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2000)*, number 1907 in Lecture Notes in Computer Science, pages 80–92, Toulouse, France, October 2000. Springer-Verlag.

[123] G. Vigna and R. A. Kemmerer. NetSTAT: A Network-based Intrusion Detection System. *Journal of Computer Security*, 7(1):37–71, 1999.

[124] G. Vigna, W. Robertson, and D. Balzarotti. Testing Network-based Intrusion Detection Systems Using Mutant Exploits. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS 2005)*, Washington DC, USA, October 2004. ACM.

[125] G. Vigna, W. Robertson, V. Kher, and R. A. Kemmerer. A Stateful Intrusion Detection System for World-Wide Web Servers. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC 2003)*, Las Vegas, NV, USA, December 2003.

[126] A. J. Viterbi. Error bounds for convolutional codes and an asymptotically optimum decoding algorithm. *IEEE Transactions on Information Theory*, 13(2):260–269, April 1967.

[127] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna. Cross Site Scripting Prevention with Dynamic Data Tainting and Static Analysis. In *Proceedings of the Network and Distributed System Security Symposium (NDSS 2007)*, February 2007.

[128] P. Wadler. The Essence of Functional Programming. In *Proceedings of the* 19^{th} *Annual Symposium on Principles of Programming Languages*, pages 1–14, Albuquerque, NM, USA, 1992. ACM.

[129] D. Wagner and D. Dean. Intrusion Detection via Static Analysis. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 2001)*, pages 156–168, Oakland, CA, USA, 2001. IEEE Computer Society.

[130] K. Wang, G. Cretu, and S. J. Stolfo. Anomalous Payload-based Worm Detection and Signature Generation. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, Seattle, WA, USA, September 2005. Springer-Verlag.

[131] K. Wang, J. J. Parekh, and S. J. Stolfo. Anagram: A Content Anomaly Detector Resistant to Mimicry Attack. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2006)*, Hamburg, GR, September 2006. Springer-Verlag.

[132] K. Wang and S. J. Stolfo. Anomalous Payload-based Network Intrusion Detection. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection (RAID 2004)*. Springer-Verlag, September 2004.

[133] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. In *Proceedings of the IEEE Symposium on Security and Privacy (S&P 1999)*, pages 133–145, Oakland, CA, USA, May 1999. IEEE Computer Society.

[134] G. Wassermann and Z. Su. Sound and Precise Analysis of Web Applications for Injection Vulnerabilities. *ACM SIGPLAN Notices*, 42(6):32–41, April 2007.

[135] G. Wassermann and Z. Su. Static Detection of Cross-Site Scripting Vulnerabilities. In *Proceedings of the 2008 International Conference on Software Engineering (ICSE 2008)*, pages 171–180, Leipzig, Germany, 2008. ACM.

[136] Y. Xie and A. Aiken. Saturn: A SAT-Based Tool for Bug Detection. In *Proceedings of the International Conference on Computer Aided Verification (CAV 2005)*, pages 139–143, July 2005.

[137] Y. Xie and A. Aiken. Static detection of security vulnerabilities in scripting languages. In *Proceedings of the USENIX Security Symposium (USENIX Security 2006)*, page 13, Vancouver, B.C., CA, August 2006. USENIX Association.

[138] D. N. Xu. Extended Static Checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, pages 48–59, Portland, OR, USA, 2006. ACM.

[139] D. N. Xu, S. P. Jones, and K. Claessen. Static Contract Checking for Haskell. In *Proceedings of the* 36^{th} *Annual ACM Symposium on the Principles of Programming Languages*, pages 41–52, Savannah, GA, USA, 2009. ACM.

[140] R. Xu and D. Wunsch. Survey of clustering algorithms. *IEEE Transactions on Neural Networks*, 16(3):645–678, 2005.

[141] S. Zanero and C. Criscione. Masibty: A Web Application Firewall based on Anomaly Detection. In *Proceedings of DeepSec – In-depth Security Conference*, November 2008.